Comparison and Implementation of Query Containment Algorithms for XPath

LINUS WÅREUS

MAX WÄLLSTEDT
Comparison and Implementation of Query Containment Algorithms for XPath

Linus Wåreus Max Wällstedt

Degree Project in Computer Science, DD143X
Supervisor: Dilian Gurov
Examiner: Örjan Ekeberg

CSC, KTH 2016-05-11
Abstract

This thesis investigates the practical aspects of implementing Query Containment algorithms for the query language XPath. Query Containment is the problem to decide if the results of one query are a subset of the results of another query for any database. Query Containment algorithms can be used for the purpose of optimising the querying process in database systems. Two algorithms have been implemented and compared, The Canonical Model and The Homomorphism Technique. The algorithms have been compared with respect to speed, ease of implementation, accuracy and usability in database systems. Benchmark tests were developed to measure the execution times of the algorithms on a specific set of queries. A simple database system was developed to investigate the performance gain of using the algorithms. It was concluded that The Homomorphism Technique outperforms The Canonical Model in every test case with respect to speed. The Canonical Model is however more accurate than The Homomorphism Technique. Both algorithms were easy to implement, but The Homomorphism Technique was easier. In the database system, there was performance to be gained by using Query Containment algorithms for a certain type of queries, but in most cases there was a performance loss. A database system that utilises Query Containment algorithms for optimisation would for every issued query have to evaluate if such an algorithm should be used.
Referat

Jämförelse och implementation av
Query Containment-algoritmer för XPath

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Purpose</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Problem Statement</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Problem Formulation</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1.1</td>
<td>Speed</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1.2</td>
<td>Ease of Implementation</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1.3</td>
<td>Accuracy</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1.4</td>
<td>Usability In Database Systems</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Scope</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Approach</td>
<td>4</td>
</tr>
<tr>
<td>1.5</td>
<td>Definitions</td>
<td>5</td>
</tr>
<tr>
<td>1.5.1</td>
<td>XML Languages and Subsets</td>
<td>5</td>
</tr>
<tr>
<td>1.5.2</td>
<td>Complexity Classes</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Background</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>The Query Containment Problem</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>XML</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>XPath</td>
<td>11</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Conjunctive Queries for XPath</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Algorithms</td>
<td>14</td>
</tr>
<tr>
<td>2.4.1</td>
<td>The Canonical Model</td>
<td>14</td>
</tr>
<tr>
<td>2.4.2</td>
<td>The Homomorphism Technique</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>Method</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Implementation</td>
<td>19</td>
</tr>
<tr>
<td>3.1.1</td>
<td>The Canonical Model</td>
<td>20</td>
</tr>
<tr>
<td>3.1.1.1</td>
<td>Correctness</td>
<td>20</td>
</tr>
<tr>
<td>3.1.2</td>
<td>The Homomorphism Technique</td>
<td>23</td>
</tr>
<tr>
<td>3.1.2.1</td>
<td>Correctness</td>
<td>23</td>
</tr>
</tbody>
</table>
3.2 Comparing Algorithms ........................................ 25
  3.2.1 Speed ................................................. 25
  3.2.2 Ease of Implementation ................................. 27
  3.2.3 Accuracy ............................................... 27
  3.2.4 Usability In Database Systems .......................... 27

4 Results 29
  4.1 Speed .................................................... 29
  4.2 Ease of Implementation .................................. 34
  4.3 Accuracy ............................................... 34
  4.4 Usability In Database Systems ............................ 34

5 Discussion 37
  5.1 The Method .............................................. 37
  5.2 The Results ............................................ 39
    5.2.1 Speed ............................................... 39
    5.2.2 Ease Of Implementation ............................ 39
    5.2.3 Accuracy .......................................... 39
    5.2.4 Usability In Database Systems ...................... 40
  5.3 The Literature .......................................... 40

6 Conclusion 41
  6.1 Conclusions ............................................ 41
  6.2 Recommendations ....................................... 42
  6.3 Future work ............................................. 43

Bibliography 45

Appendices 46

A Code 47

B Grammars 49
  B.1 XML .................................................... 49
  B.2 XPath .................................................. 49

C Ease of Implementation Questionnaire 51
  C.1 Template .............................................. 51
  C.2 The Canonical Model ................................... 52
  C.3 The Homomorphism Technique ............................ 53
Chapter 1

Introduction

In this thesis, the Query Containment problem is studied. Query Containment is the problem to decide whether the results of one query are contained in the results of another query for a database [1, 4].

The abstraction of databases that is mostly used today is the relational database. It was theorised in the seventies, and has since been thoroughly studied from a logical perspective, introducing the field of Database Theory [1]. Database systems have since evolved and grown more complex as the amount of data to be stored has increased. Optimising querying over databases has therefore become more relevant. Over the last few decades, a method for optimising database querying that has been studied is Query Containment [1, 3], which is the subject for this thesis.

The general Query Containment problem has been proven to be undecidable for most query languages, including relational algebra, tuple calculus and SQL [1, 8]. This means that no algorithm can solve the general Query Containment problem. A smaller and restricted query language must therefore be used to make the problem decidable. The typical restricted query language used for Query Containment is Conjunctive Queries. This query language is still expressive enough for most practical uses. It is also common to extend Conjunctive Queries with different operations, while retaining decidability, to make the language more expressive [1, 5].

For the purpose of this thesis, the database architecture that has been chosen is XML. XML is a structured markup language used to store structured data in plain text [2]. XML is used in many different applications, e.g. to send data over the Internet because of its simplicity. The main language used to query an XML document for data is XQuery [9]. XPath is a subset of XQuery that navigates the nodes of an XML document and returns the specified nodes [11]. XML Queries are relatively new in database theory, but have been studied to some extent with respect to Query Containment, e.g. [3, 4, 10].
1.1 Purpose

The purpose of this thesis is to study practical aspects of Query Containment for XPath. There have been several methods proposed to determine Query Containment for XPath [10]. This thesis compares some of these methods to find their strengths and weaknesses and finds conclusions regarding which algorithm is preferable in which context.

Another purpose of this thesis is to study implementation aspects of the Query Containment problem. The Query Containment problem is often described in a theoretical context and neglects the practical aspects of implementation. Query Containment algorithms are therefore implemented to study this aspect.

The results of this thesis can be used for optimising database systems in different respects. For example, the caching process can be made more efficient in database systems by using the most appropriate algorithm for the context [3]. The results can also be used when optimising a query before issuing the query on a database.

Another area where Query Containment can be useful is to reduce network bandwidth, especially for mobile devices. If a device is planning to send a query to a server, it can instead test if the query is contained in another query that was previously sent to the server. The result from the previous query can then be cached and reused, without sending a new request to the server over the network [6].
1.2. PROBLEM STATEMENT

1.2 Problem Statement

The goal of this thesis is to find and implement Query Containment algorithms and compare their strengths and weaknesses in different respects.

1.2.1 Problem Formulation

The problem statement in this thesis is as follows:

*Which of the selected algorithms studied in this thesis has the best performance with respect to the following criteria:*

1. *Speed*
2. *Ease of implementation*
3. *Accuracy*
4. *Usability in database systems*

1.2.1.1 Speed

A fast algorithm can be used in a system where speed is critical and inaccuracies do not matter too much. An algorithm for a smaller subset of the query language can be faster than a more comprehensive algorithm for a greater subset of the query language. The fastest algorithm might therefore depend on which subset of the query language that is used.

1.2.1.2 Ease of Implementation

The easiest algorithm to implement is probably not the best in any other respect, since good algorithms are in the authors’ experience typically more complex than inferior algorithms. It can however be good to have an easy algorithm at hand in some cases for quick access and implementation.

1.2.1.3 Accuracy

An accurate algorithm will always give an exact answer for any input of its subset of the query language. A less accurate algorithm might give false results in certain contexts. The algorithm that has the most accurate results will also probably be the slowest, especially for large input data. This is because of the high complexity of the problem itself. For small queries however, an accurate algorithm can be quick enough and will then provide accurate results.
1.2.1.4 Usability In Database Systems

An algorithm that is usable in database systems needs to take every respect into consideration. Finding only correct results is in some cases too slow, so a database system might rather use an algorithm that does not find only correct results, but at least a sufficient ratio of correct results. In some cases, the algorithm will not speed up the querying process at all, but instead slow it down because the Query Containment algorithm was not fast enough. This contradict the purpose of using such an algorithm, since the goal of using a Query Containment algorithm, in this case, is to speed up the querying run time.

1.3 Scope

The scope of this thesis includes implementing Query Containment algorithms for subsets of XPath and to compare them. No other query languages, such as SQL, are investigated, since XML databases and full feature relational databases are very different, and not very meaningful to compare to each other.

The aim is not to develop new algorithms that solve the Query Containment problem, but rather to investigate some of the algorithms that are already proposed. The algorithms that are implemented and compared are:

- The Canonical Model
- The Homomorphism Technique

1.4 Approach

The goal of this thesis is, as previously mention, to implement different Query Containment algorithms and to investigate which ones are the most useful, given the criteria in section 1.2.1. This is done using benchmark tests, a questionnaire and a literature study for comparing different qualities for the algorithms. The benchmark tests compare the execution time of an algorithm, given different sizes for the input data. The measured speeds of the algorithms are used to determine which algorithm that fulfils criterion 1, speed. Further details in section 1.2.1.1. The benchmark results are also compared to the time complexity of the algorithms. The questionnaire is used to determine which algorithm that fulfils criterion 2, ease of implementation. Further details in section 1.2.1.2. The literature study is used to determine which algorithm that fulfils criterion 3, accuracy. Further details in section 1.2.1.3. Criterion 4, usability in database systems, is determined by implementing a simple database system and comparing the execution time of using an algorithm against not using one. Further details in section 1.2.1.4.
## 1.5 Definitions

### 1.5.1 XML Languages and Subsets

<table>
<thead>
<tr>
<th>Term</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XPath</td>
<td>XML Path Language</td>
</tr>
<tr>
<td>XQuery</td>
<td>XML Query Language</td>
</tr>
</tbody>
</table>

### 1.5.2 Complexity Classes

<table>
<thead>
<tr>
<th>Term</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>Polynomial Time</td>
</tr>
<tr>
<td>NP</td>
<td>Non-deterministic Polynomial Time</td>
</tr>
<tr>
<td>coNP</td>
<td>Complement of NP</td>
</tr>
<tr>
<td>PSPACE</td>
<td>Polynomial Space</td>
</tr>
<tr>
<td>$\Pi_2^P$</td>
<td>The second level of the polynomial hierarchy (NP is the first level)</td>
</tr>
<tr>
<td>EXPTIME</td>
<td>Exponential Time</td>
</tr>
<tr>
<td>NEXPTIME</td>
<td>Non-deterministic Exponential Time</td>
</tr>
<tr>
<td>coNEXPTIME</td>
<td>Complement of NEXPTIME</td>
</tr>
</tbody>
</table>
Chapter 2

Background

In this chapter, the theoretical background necessary for this thesis is presented in detail. The different algorithms that are studied in this thesis are explained in depth and some other related technologies are also presented.

2.1 The Query Containment Problem

The Query Containment problem is theorised as deciding whether

\[ Q_1(D) \subseteq Q_2(D) \]

for two queries \( Q_1, Q_2 \) and a database \( D \). Generally a query on a database return a set of tuples. The Query Containment problem determines if

\[ t \in Q_1(D) \rightarrow t \in Q_2(D) \]

for a tuple \( t \), i.e. if every possible tuple in \( Q_1(D) \) by necessity also is a tuple in \( Q_2(D) \).
Chapter 2. Background

An example of containment in bag semantics is:

\[ D(a, b, c) = \{(1, 1, 1), (1, 1, 2), (1, 2, 3)\} \]

\[ Q_1(R) = \pi_{a, b}(\sigma_{a=1 \land b=1}(R)) \]
\[ Q_2(R) = \pi_{a, b}(\sigma_{a=1}(R)) \]

\[ Q_1(D) = \{(1, 1), (1, 1)\} \]
\[ Q_2(D) = \{(1, 1), (1, 1), (1, 2)\} \]

In table 2.1-2.3, the equivalent example in SQL is given. The tuples in \( Q_2(D) \) are the tuples in \( D \) with \( a = 1 \) projected on \( (a, b) \), and every tuple in \( Q_1(D) \) are the tuples in \( D \) with \( a = 1 \land b = 1 \) projected on \( (a, b) \). \( Q_1 \) is more restrictive than \( Q_2 \), and they share the same projection and \( a = 1 \). Every tuple in \( Q_1 \) is therefore contained in \( Q_2 \), and \( Q_1(D) \subseteq Q_2(D) \) is true by necessity.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: SELECT * FROM D

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: SELECT a, b FROM D WHERE a = 1 AND b = 1

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.3: SELECT a, b FROM D WHERE a = 1
2.1. THE QUERY CONTAINMENT PROBLEM

The Query Containment problem is known to be undecidable for most query languages, including relational algebra, tuple calculus, SQL and XPath [1, 8, 10]. In order to make Query Containment decidable, the query language needs to be limited. A typical subset of queries used in Query Containment calculations is Conjunctive Queries [1, 8].

Conjunctive Queries are queries that can be written as a tuple calculus expression only using two operations, conjunction, $\land$, and existential quantification, $\exists$. Equality, $=$, is the only comparison operator that can be used. For example:

$$\{(x.student, x.address) \mid \text{Lives}(x) \wedge x.city = 'Stockholm' \wedge (\exists y) (\text{Attends}(y) \wedge y.course = 'DD143X') \wedge x.student = y.student\}$$

This is the equivalent of queries only using selection, $\sigma$, projection, $\pi$, and cartesian product, $\times$, operators. For example:

$$A := \text{Attends}$$
$$L := \text{Lives}$$
$$S := \sigma_{A.student=\text{L.student}}(\sigma_{\text{course}=\text{DD143X}}(A) \times \sigma_{\text{city}='Stockholm'}(L))$$
$$\text{Res} := \pi_{\text{L.student, L.address}}(S)$$

Query Containment for Conjunctive Queries has been proven to be NP-complete. NP-complete for Query Containment is considered acceptable, since queries on a database are typically small in size [1, 5].

When Conjunctive Queries is not enough, they are extended to include some more operators. This can be done in a way so that Query Containment is still decidable. Conjunctive Queries can be extended with the comparison predicates inequality, $\neq$, less than, $<$, and less than or equal to, $\leq$. This extension of Conjunctive Queries is $\Pi_2^p$-complete [5]. This is also true for just the inequality predicate. This complexity class is considered a borderline case for solving the Query Containment problem in reasonable time. The previous extensions to Conjunctive Queries all assume set semantics. In bag semantics, the Query Containment problem is undecidable for Conjunctive Queries with inequalities [5].
CHAPTER 2. BACKGROUND

2.2 XML

XML documents encode structured data as plain text using tags [2], e.g. `<name>John</name>`. Tags can be nested, yielding a tree structure. For the purpose of thesis, XML will therefore be considered unranked trees. In figure 2.1, an example of an XML document and its equivalent tree is given. XML is used for several purposes. A large area that XML is used in is to send data over the Internet [2]. This is done because HTTP-requests are encoded in plain text, and it is natural to also transmit the data in these requests as plain text.

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

**Figure 2.1:** Example of an XML document and its corresponding tree.
2.3 XPath

To select specific nodes from an XML document, XPath is mostly used as the query language [11]. XPath is a string that describes the steps and conditions of the nodes in question. Consider the XPath expression \( p = //c//\ast \). If \( p \) is issued on the XML database in figure 2.1, the result would be:

\[
\begin{align*}
&<c><b></b></c> \\
&<c><c></c></c>
\end{align*}
\]

The XPath expression \( p \) searches for any \( c \)-node in the XML document that has a child node of any kind. Query Containment with respect to XPath is the problem to decide whether every possible node in the results of an XPath expression \( p \) also is in the results of an XPath expression \( q \) by necessity.

For the purpose of this thesis, subsets of XPath are studied. The notation for XPath subsets used in this thesis uses the following operators [11]:

- / is a child step.
- // is a descendant step.
- \( \ast \) is a wildcard node.
- [] is a branching predicate.

A child step \( a/b \) indicates that a node \( b \) must be directly under a node \( a \) in the XML tree. A descendant step \( c//d \) indicates that a node \( d \) must be somewhere under a node \( c \) in the XML tree. A wildcard node \( \ast \) indicates that the node can have any label. A branching predicate \( a[p] \) indicates that the XPath expression \( p \) must be true in relation to a node \( a \). Consider the XPath expression \( a[//c]/b \). This means that a node \( a \) must have a descendant node \( c \) and a child node \( b \), and it is the \( b \)-node that is returned.

A subset of XPath that only uses child steps and wildcard nodes is referred to as \( XPath(//, \ast) \). A list of the complexity classes of the Query Containment problem for different XPath subsets can be found in table 2.4. More details about the constraints in the XPath subsets can be found in [10].
### Complexity Class | XPath Subset
---|---
**PTIME** | $\text{XPath}(//, [\cdot], \ast)$  
| $\text{XPath}(//, [\cdot], \ast)$  
| $\text{XPath}(//, [\cdot], [\cdot])$ with fixed bounded SXICs  
| $\text{XPath}(//) + \text{DTDs}$  
| $\text{XPath}([\cdot]) + \text{DTDs}$  
**coNP** | $\text{XPath}(//, [\cdot], \ast)$  
| $\text{XPath}(//, [\cdot], \ast, [\cdot]), \text{XPath}(//, [\cdot]), \text{XPath}(//, [\cdot])$  
| $\text{XPath}(//, [\cdot]) + \text{DTDs}$  
| $\text{XPath}([\cdot], [\cdot]) + \text{DTDs}$  
**$\Pi_2^p$** | $\text{XPath}(//, [\cdot], [\cdot], [\cdot], [\cdot])$  
| + existential variables  
| + path equality  
| + ancestor-or-self axis  
| + fixed bounded SXICs  
| $\text{XPath}(//, [\cdot], [\cdot], [\cdot], [\cdot])$  
| + existential variables  
| + all backward axes  
| + fixed bounded SXICs  
| $\text{XPath}(//, [\cdot], [\cdot], [\cdot])$  
| + existential variables with inequality  
**PSPACE** | $\text{XPath}(//, [\cdot], [\cdot], [\cdot], [\cdot]), \text{XPath}(//, [\cdot], [\cdot], [\cdot], [\cdot])$ if the alphabet is finite  
| $\text{XPath}(//, [\cdot], [\cdot], [\cdot], [\cdot])$  
| + variables with XPath semantics  
**EXPTIME** | $\text{XPath}(//, [\cdot], [\cdot], [\cdot], [\cdot])$  
| + existential variables  
| + bounded SXICs  
| $\text{XPath}(//, [\cdot], [\cdot], [\cdot], [\cdot])$  
| + DTDs  
| $\text{XPath}(//, [\cdot], [\cdot]) + \text{DTDs}$  
| $\text{XPath}([\cdot], [\cdot], [\cdot], [\cdot]) + \text{DTDs}$  
**Undecidable** | $\text{XPath}(//, [\cdot], [\cdot], [\cdot], [\cdot])$  
| + existential variables  
| + unbounded SXICs  
| $\text{XPath}(//, [\cdot], [\cdot], [\cdot], [\cdot])$  
| + existential variables  
| + bounded SXICs  
| $\text{XPath}(//, [\cdot], [\cdot], [\cdot], [\cdot])$  
| + DTDs  
| $\text{XPath}(//, [\cdot], [\cdot], [\cdot], [\cdot])$  
| + nodeset equality  
| + simple DTDs  
| $\text{XPath}(//, [\cdot], [\cdot], [\cdot], [\cdot])$  
| + existential variables with inequality

**Table 2.4:** Complexity classes of the Query Containment problem for different XPath classes [10].
2.3. XPATH

Consider an example with two XPath expressions $p = //c[/*]$ and $q = //c$ issued on the XML database in figure 2.1. The expression $p$ selects all nodes of the database with the label $c$ that also have a child node with any label. The query would return

$$<c><b></b></c>
<c><c></c></c>$$

The expression $q$ selects all nodes with the label $c$ in the database with no further restrictions. The query would return

$$<c></c>
<c><b></b></c>
<c><c></c></c>
<c></c>$$

Since every node selected by $p$ also will be selected by $q$ regardless of the contents of the database, Query Containment $p \subseteq q$ is true.

2.3.1 Conjunctive Queries for XPath

XQuery is typically the query language used for querying an XML database [4]. Since the general Query Containment problem for XQuery is undecidable [10], a subset of XQuery that is equivalent of Conjunctive Queries is used, which is called c-XQuery. In c-XQuery, the subset of XPath that is used is $XPath(\text{/}, [], *)$ [4]. As can be seen in table 2.4, the complexity class of the Query Containment problem for $XPath(\text{/}, [], *)$ is PTIME. It is therefore natural to extend the XPath subset in the Query Containment algorithms and use an extension of Conjunctive Queries.
CHAPTER 2. BACKGROUND

2.4 Algorithms

The algorithms studied in this thesis consider an XPath expression a tree. Consider \( p_0 = a[/d]*//c \). Let \( T(p) \) be the tree representation of an XPath expression \( p \). \( T(p_0) \) can be seen in figure 2.2. The tree selects nodes under the root node with label \( a \) so that \( a \) has a descendant node \( d \), a wildcard child node that in turn has a descendant node \( c \), and these \( c \)-nodes are returned [10].

\[ a \]
\[ \text{d} \quad \star \]
\[ \text{c} \]

Figure 2.2: The tree \( T(p_0) \) corresponding to the XPath expression \( p_0 \).

2.4.1 The Canonical Model

The Canonical Model is a technique to test Query Containment for XPath expressions \( p, q \) by finding a counter example \( p' \) where \( q \neq p' \). This is done by transforming \( p \) according to the following two rules:

1. Replace every wildcard, \( * \), with a label \( z \) so that \( z / \notin T(p) \).
2. Replace every descendant edge, \( // \), with a chain of at most \( m(q) + 1 \) child edges and nodes labelled by \( z \), where \( m(q) \) is defined as the length of the longest chain of child edges and wildcard nodes.

If \( q \) matches every transformed tree from \( p \), no counter example was found and \( p \subseteq q \). The subset of XPath that the algorithm solves is \( XPath(//, //, [], *) \) [10].

Consider an example from [10] of a run of The Canonical Model on two queries \( p_1 = a/b[/d]*//c \) and \( q_1 = a[/d]/*//c \). The tree representations of the expressions are given in figure 2.3. The longest child to wildcard chain in \( q_1 \) is 1, i.e. \( m(q_1) = 1 \). The transformations of \( p_1 \) are given in figure 2.4. To determine if \( q_1 \) matches all of the transformed trees, the transformations can be considered XML trees. The XPath expression \( q_1 \) is issued on all these XML trees. If the query returns nodes on every transformation, \( p_1 \subseteq q_1 \) is true. In this example, it is evident that \( q_1 \) matches every transformed tree, and there is containment.
2.4. ALGORITHMS

Figure 2.3: Tree representations of the XPath expressions $p_1$ and $p_2$

Figure 2.4: Transformations of $p_1$
2.4.2 The Homomorphism Technique

The Homomorphism Technique proves Query Containment \( p \subseteq q \) by finding a homomorphism \( h \) from \( T(q) \) to \( T(p) \), where \( T(expr) \) is the equivalent tree structure of an XPath expression \( expr \). The homomorphism is constructed according to the following rules:

1. The root node of \( T(q) \) must be mapped to the root node of \( T(p) \).
2. If \( (u, v) \) is a child edge in \( T(q) \), \( (h(u), h(v)) \) must be a child edge in \( T(p) \).
3. If \( (u, v) \) is a descendant edge in \( T(q) \), \( h(v) \) must be under \( h(u) \) in \( T(p) \).
4. If \( u \) has a label \( e \neq \ast \), \( h(u) \) must also have the label \( e \).

If no homomorphism is found, \( p \nsubseteq q \) follows. This algorithm solves Query Containment for \( XPath(/, //, []) \) and \( XPath(/, [], \ast) \), but not entirely for \( XPath(/, //, [], \ast) \), since the existence of a homomorphism is not necessary for containment in that case [1, 10].

As an example, consider the expressions \( p_1 \) and \( q_1 \) in section 2.4.1. The equivalent tree structures \( T(p_1) \) and \( T(q_1) \) are given in figure 2.3. The first mapping rule to satisfy is that the root node of \( T(q_1) \) must be mapped to the root node of \( T(p_1) \). Rule 4 is also satisfied, since both nodes have the label \( \text{a} \). This mapping is shown in figure 2.5. This means that \( h(T(q_1).a) = T(p_1).a \). \( T(p).n \) refers to the node with the label \( \text{n} \) in the tree structure of the XPath expression \( p \).

In the next step, consider the descendant \( \text{d} \)-node under the \( \text{a} \)-node in \( T(q_1) \). Rule 3 says that \( h(T(q_1).d) \) must be somewhere under \( h(T(q_1).a) \) in \( T(p_1) \). There is only one \( \text{d} \)-node in \( T(p_1) \), and it is indeed under \( T(p_1).a \), so the rule is satisfied, and the mapping is shown in figure 2.6.

For the next mapping, consider the \( \ast \)-node in \( T(q_1) \). According to rule 4, it can be mapped to a node of any label, and according to rule 2, it must be mapped to a child node of \( h(T(q_1).a) \). There is only one child node of \( h(T(q_1).a) \), which means that \( h(T(q_1).\ast) = T(p_1).b \). This mapping can be seen in figure 2.7.

The last node to map in \( T(q_1) \) is the \( \text{c} \)-node. It has to be mapped to a node under \( h(T(q_1).\ast) \) with the label \( \text{c} \). There is such a node in \( T(p_1) \), which means that \( h(T(q_1).c) = T(p_1).c \). All the nodes in \( T(q_1) \) are now mapped to \( T(p_1) \) in a way that satisfies the rules, which means that there is a homomorphism from \( q \) to \( p \), which implies Query Containment \( p \subseteq q \).
2.4. ALGORITHMS

Figure 2.5: The mapping from $T(q_1).a$ to $T(p_1).a$

Figure 2.6: The mapping from $T(q_1).d$ to $T(p_1).d$

Figure 2.7: The mapping from $T(q_1).*$ to $T(p_1).b$

Figure 2.8: The mapping from $T(q_1).c$ to $T(p_1).c$
Chapter 3

Method

In this chapter, the method of this thesis is presented in detail to explain how the algorithms are implemented and compared according to the criteria in section 1.2.1. The benchmark tests will also be presented in detail to explain how they are implemented and used.

3.1 Implementation

Developing a Query Containment algorithm consists of a few steps. A parser for the query language must be developed, the subset of the query language that an algorithm can handle must be confirmed, and the algorithm itself must be developed with respect to the syntax tree generated by the parser. The grammars used in the parsers can be found in appendix B.

The syntax tree for the relevant subsets of XPath is constructed through recursive descent. A tree node is either a child node or a descendant node. Every tree node has a list of child and descendant nodes. Examples of this syntax tree can be found in figure 3.1.

(a) /a[//d]/*//c 
(b) /a[//d//e[//b]/*//c/d

Figure 3.1: The syntax tree for the relevant subsets of XPath.
3.1.1 The Canonical Model

The Canonical Model is a containment algorithm for XPath(/, //, [, *, ). It is described in detail in section 2.4.1. The pseudo code for the implementation of the algorithm can be found in algorithm 1 and an implementation in C of this algorithm can be found in appendix A.

Considering two XPath expressions \( p, q \) and the containment test \( p \subseteq q \), transforming the tree \( p \) is done by first replacing every wildcard node, \(*\), with a label \( z \) so that \( z \not\in T(p) \). The second step is to generate new trees where every descendant edge has been replaced with a chain of at most \( m(q) + 1 \) child nodes with the label \( z \) [10]. This process is shown in figure 3.2.

As can be seen in figure 3.2, the amount of trees generated depends on the number of descendant edges in \( p \) and \( m(q) \). If we let \( d(p) \) be the number of descendant edges in \( p \), the exact number of trees generated is \( (m(q) + 2)^{d(p)} \). The complexity is exponential with respect to \( d(p) \). For every generated tree \( p' \), the algorithm tests if \( q \) matches \( p' \). The generated tree is treated as an XML tree and the matching is done by testing if \( q \) gives any results for the generated tree.

3.1.1.1 Correctness

An important aspect of implementing an algorithm is to ensure its correctness. To determine the correctness of The Canonical Model, the steps in section 2.4.1 have to be considered. The first step in the algorithm is to replace every wildcard node in \( p \) with a new label \( z \). In the pseudo code in algorithm 1, this is done on line 4. The second step is to replace the descendant edges in \( p \) with a chain of at most \( m(q) + 1 \) child-to-\( z \) steps. In the implementation, this is done by getting references to all descendant edges in \( p \) on line 5, and inserting all possible combinations of such chains in the loop on line 12. \( m(q) \) is calculated on line 2. In the algorithm, \( q \) must match every transformed tree. This is done on line 23. This concludes that all the steps of the algorithm are performed, and the implementation of the algorithm is correct.
3.1. IMPLEMENTATION

(a) \( p = \text{root}/a///a[/b] \)

(b) \( p^{(1)} \)

(c) \( p^{(2)} \)

(d) \( p^{(3)} \)

(e) \( p^{(4)} \)

(f) \( p^{(5)} \)

(g) \( p^{(6)} \)

(h) \( p^{(7)} \)

(i) \( p^{(8)} \)

(j) \( p^{(9)} \)

Figure 3.2: The descendant edge replacement process of The Canonical Model with \( m(q) + 1 = 2 \).
Algorithm 1 Pseudo code for The Canonical Model

1: function TheCanonicalModel(XPath Tree p, XPath Tree q)
2:   \( m_q \leftarrow \text{LongestChildWildcardChain}(q) \)
3:   \( z \leftarrow \text{NewLabelNotInTrees}(p, q) \)
4:   \( p \leftarrow \text{XPathTreeReplaceWildcards}(p, z) \)
5:   \( (d_{\text{from}}[], d_{\text{to}}[]) \leftarrow \text{XPathTreeGetDescendantEdges}(p) \)
6:   \( \mathit{it}[] \leftarrow \text{ARRAY}(|d_{\text{from}}|) \)
7:   \( \mathit{it}[] \leftarrow 0 \)
8:   for \( i \leftarrow 0 \) to \( |\mathit{it}| - 1 \) do
9:     \( \mathit{it}[i] \leftarrow 0 \)
10: end for
11: for \( i \leftarrow 0 \) to \((m_q + 2)|\mathit{it}| - 1\) do
12:   for \( j \leftarrow 0 \) to \( |\mathit{it}| - 1 \) do
13:     if \( \mathit{it}[j] = 0 \) then
14:       \( d_{\text{from}}[j].\text{next} \leftarrow d_{\text{to}}[j] \)
15:     else
16:       \( (t_{\text{first}}, t_{\text{last}}) \leftarrow \text{NewXPathTreeChain}(z, \mathit{it}[j]) \)
17:       \( d_{\text{from}}[j].\text{next} \leftarrow t_{\text{first}} \)
18:       \( t_{\text{last}}.\text{next} \leftarrow d_{\text{to}}[j] \)
19:     end if
20:   end for
21: end for
22: if TreesMatch(p, q) \neq \text{true} then
23:   return false
24: end if
25: for \( j \leftarrow 0 \) to \( |\mathit{it}| - 1 \) do
26:   \( \mathit{it}[j] \leftarrow \mathit{it}[j] + 1 \)
27:   if \( \mathit{it}[j] < m_q + 2 \) then
28:     break
29:   else
30:     \( \mathit{it}[j] \leftarrow 0 \)
31:   end if
32: end for
33: return true
34: end function
3.1. IMPLEMENTATION

3.1.2 The Homomorphism Technique

The Homomorphism Technique is a containment algorithm for $XPath(/, [], *)$, $XPath(/, //, [])$ and to some extent $XPath(/, //, [], *)$. It is described in detail in section 2.4.2. The pseudo code for the implementation can be found in algorithm 2 and an implementation in C of this algorithm can be found in appendix A.

The algorithm determines Query Containment $p \subseteq q$ by finding a homomorphism from $T(q)$ to $T(p)$ [10]. The algorithm’s structure makes it natural to implement it recursively, since it starts at the root node of $T(q)$ and satisfies the rules for every edge in the tree.

The algorithm starts with comparing the labels of the root nodes of $T(p)$ and $T(q)$. The next step is to ensure that all child edges in $T(q)$ has a homomorphism with one of the child edges in $T(p)$. The special case is the descendant edges in $T(q)$. They have to be mapped to any edge under $T(p)$. This is done by a separate function.

3.1.2.1 Correctness

The correctness of the implementation of The Homomorphism Technique is easiest explained by first disregarding descendant edges. In that case, the current node of $q$ must either be a wildcard or have the same label as $p$. Every child node in $q$ must have an equivalent child node in $p$, which means that the algorithm can recursively find a homomorphism from every child node in $q$ to a child node in $p$. In the pseudo code in algorithm 2, this is done in the function TheHomomorphismTechnique on line 1. First an outer loop iterates over all child edges in $q$ on line 6, and then an inner loop iterates over all child edges in $p$ on line 9, making a recursive call to find a homomorphism. If a homomorphism can be found for every child edge, the algorithm returns true.

Descendant edges are handled by the function DescendantHomomorphism on line 29 that is called for every descendant edge in $q$ on line 21. A descendant edge in $q$ means that the node must be mapped to any node under $p$. The function tries to map the descendant node in $q$ to every child and descendant node of $p$ by calling the main function TheHomomorphismTechnique. If it cannot be mapped, the function DescendantHomomorphism is called recursively on the child or descendant node of $p$. As soon as the descendant node of $q$ could be mapped to any node under $p$, DescendantHomomorphism terminates and returns true. If no mapping could be done, the function returns false, which in turn means that the entire algorithm returns false. This concludes that all the rules of The Homomorphism Technique in section 2.4.2 have been satisfied, and the implementation of the algorithm is correct.
Algorithm 2 Pseudo code for The Homomorphism Technique

1: function TheHomomorphismTechnique(XPath Tree \( p \), XPath Tree \( q \))
2:    if \( q.\text{label} \neq * \land q.\text{label} \neq p.\text{label} \) then
3:        return false
4:    end if
5:    for all child edges \( c_q \) in \( q \) do
6:        edge\_ok \( \leftarrow \) false
7:    end for
8:    for all child edges \( c_p \) in \( p \) do
9:        if TheHomomorphismTechnique\((c_p, c_q)\) = true then
10:           edge\_ok \( \leftarrow \) true
11:        end if
12:    end for
13:    if edge\_ok \( \neq \) true then
14:        return false
15:    end if
16:    for all descendant edges \( d_q \) in \( q \) do
17:        if DescendantHomomorphism\((p, d_q)\) \( \neq \) true then
18:            return false
19:        end if
20:    end for
21:    return true
22: end function

23: function DescendantHomomorphism(XPath Tree \( p \), XPath Tree \( q \))
24:    for all edges \( e_p \) in \( p \) do
25:        if TheHomomorphismTechnique\((e_p, q)\) = true then
26:            return true
27:        end if
28:        if DescendantHomomorphism\((e_p, q)\) = true then
29:            return true
30:        end if
31:    end for
32:    return false
33: end function
3.2 Comparing Algorithms

The algorithms are compared to each other with respect to the criteria in section 1.2.1. In the following sections, the comparison method for every criterion is explained.

3.2.1 Speed

The speed of the algorithms is measured with benchmark tests. Every algorithm runs the same test and their average execution time is compared to the other algorithms in a graph. A specific set of test queries has been developed to cover different aspects of XPath expressions. These test cases can be found in table 3.1.

The benchmark tests measure the run time of a query in approximated CPU time, using the clock_t clock(void); function from time.h in the C standard library. A code segment with the structure of the benchmark tests is given in listing 3.1. The benchmarks are executed 10 000 times, and an average time is used in the comparison to eliminate fluctuations. The measured times are compiled in graphs, using GNU Octave. In the graphs, the algorithms can be compared for every test case. The code for the benchmark tests can be found in appendix A.

```c
#include <time.h>
clock_t start, end;
start = clock();
/* Perform the algorithm */
end = clock();
return end - start;
```

**Listing 3.1:** A code segment for measuring execution time of an algorithm.
<table>
<thead>
<tr>
<th>Query Form</th>
<th>XPath Subset</th>
<th>Test Case for the Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>//</strong></td>
<td>/root</td>
<td>1. Child branch (depth 2)</td>
</tr>
</tbody>
</table>
3.2. COMPARING ALGORITHMS

3.2.2 Ease of Implementation

Measuring the ease of implementing an algorithm can be done in different ways, with varying exhaustiveness. For the purpose of this thesis, it has been decided that primarily two aspects should be investigated and measured, namely development time and the subjective opinion of the implementer. A questionnaire has been developed to streamline this process, and it can be found in appendix C. The questionnaire consists of 4 questions, where every question asks for a rating on a scale of five levels. The first alternative means that the algorithm was very easy or fast to implement, with respect to the question asked and the last alternative means that the algorithm was very hard or time consuming to implement, with respect to the question asked. An average grade is computed and then used to compare this criterion for the different algorithms. It should be noted that both development time and the subjective opinion of the implementer will vastly vary from implementer to implementer. This criterion should therefore be the least important one to consider, and is rather meant for giving some idea to a reader who would want to further study and implement the algorithms themselves.

3.2.3 Accuracy

Since none of the algorithms are approximations or heuristics, the rate of false results is not measured. Instead, the subsets of XPath that the algorithms can use effectively are compared in a literature study, with the primary source of information being [10].

3.2.4 Usability In Database Systems

A small database system has been developed to measure if the usage of a Query Containment algorithm actually makes the system more efficient. The system takes an XML document, two XPath expressions and a Query Containment algorithm as input, and outputs the results of the queries. The system performs the specified Query Containment algorithm before fetching the results from the database to see if there is containment between the two queries. If there is no containment, both queries will be issued as normal on the database. If there is containment, the bigger query will be issued first on the database, and the smaller contained query will then fetch its results from the results of the bigger query.

The system consists of four main parts. The first one is the parsers, that reads the XML file and XPath expressions and constructs their equivalent syntax trees. The second part is the querying process, that picks out all the specified nodes of the database to the results of a query. The third part is the Query Containment tester, which tests Query Containment between the queries with the specified algorithm. The last part is the result fetching process, which fetches the results of a contained query from the results of the bigger query.
The usability in database systems of the algorithms is measured with benchmark tests. For every test in table 3.1, an equivalent test has been constructed for a real XML database. The database used in this thesis is `mondial.xml` [7]. Every algorithm is compared to not using an algorithm at all. The code for the database system can be found in appendix A.
Chapter 4

Results

In this chapter, the results for every criterion in section 1.2.1 of the thesis is presented.

4.1 Speed

The speed of the algorithms was measured using benchmark tests on the queries in table 3.1. Every test was performed 10 000 times, and the average time was used.

Figure 4.1 compares the algorithms on test 1, *Child*. Chains up to a length of 10 steps were tested.

Figure 4.2 compares the algorithms on test 2, *Descendant*. Chains up to a length of 10 steps were tested.

Figure 4.3 compares the algorithms on test 7, *Child branch (depth 2)*. Ranks up to 10 were tested.

Figure 4.4 compares the algorithms on test 8, *Descendant branch (depth 2)*. Ranks up to 3 were tested.

Figure 4.5 compares the algorithms on test 13, *Child branch (rank 2)*. Depths up to 10 levels were tested.

Figure 4.6 compares the algorithms on test 14, *Descendant branch (rank 2)*. Depths up to 3 levels were tested.
It is evident that The Homomorphism Technique performs better than The Canonical Model in all of these tests, especially with descendant steps. The tests in table 3.1 that are not presented in this chapter have been omitted for two reasons. Wildcards did not affect the results significantly, so the graphs appears too similar to the ones without the wildcards, and the results do not provide any new information. The other tests that are omitted are the ones with both child and descendant steps. These tests behave as one would expect by studying the separate child and descendant graphs. The graphs for the omitted tests can be found in appendix A.
4.1. SPEED

**Figure 4.1:** Execution times for test 1. *Child*

![Figure 4.1: Execution times for test 1. *Child*](image1)

**Figure 4.2:** Execution times for test 2. *Descendant*

![Figure 4.2: Execution times for test 2. *Descendant*](image2)
**Figure 4.3:** Execution times for test 7. *Child branch (depth 2)*

**Figure 4.4:** Execution times for test 8. *Descendant branch (depth 2)*
4.1. SPEED

**Figure 4.5:** Execution times for test 13. *Child branch (rank 2)*

![Graph showing execution times for test 13.](image)

**Figure 4.6:** Execution times for test 14. *Descendant branch (rank 2)*

![Graph showing execution times for test 14.](image)
4.2 Ease of Implementation

The results of the questionnaires can be found in appendix C. The Canonical Model was assigned a score of 10 and an average score of 2.5, as can be seen in appendix C.2. The Homomorphism Technique was assigned a score of 5 and an average score of 1.25, as can be seen in appendix C.3. If these average scores are rounded to the nearest integer 3 and 1 respectively, it means that The Canonical Model had a difficulty of normal and The Homomorphism Technique had a difficulty of very easy, on a scale of very easy, easy, normal, hard and very hard.

4.3 Accuracy

The Canonical Model is accurate on XPath(/, //, [ ], *), as previously stated in section 2.4.1, whilst The Homomorphism Technique is accurate on XPath(/, //, [ ]) and XPath(/, [ ], *), as previously stated in section 2.4.2. The Homomorphism Technique can give false negative results on XPath(/, //, [ ], *), i.e. the algorithm can return no containment even if there actually was containment. This means that The Canonical Model is the more accurate algorithm.

4.4 Usability In Database Systems

The usability in database systems was measured using benchmark tests on the querying process when two queries were issued on a big database, namely the database mondial.xml [7]. The execution times of using the different algorithms were compared to not using an algorithm at all. Equivalent tests to the ones in table 3.1 were used. Every test was performed 100 times and the average time was used.

Figure 4.7 compares the algorithms on test 3, Child descendant. Chains up to a length of 4 steps were tested.

Figure 4.8 compares the algorithms on test 9, Child descendant branch (depth 2). Ranks up to 4 were tested.

Figure 4.9 compares the algorithms on test 15, Child descendant branch (rank 2). Depths up to 4 levels were tested.
4.4. USABILITY IN DATABASE SYSTEMS

By analysing the graphs, it is evident that not using an algorithm outperforms using one in most cases. The exceptions are the ones only using child and descendant steps in figure 4.7. The tests in table 3.1 that are not presented in this chapter are omitted for similar reasons as in section 4.1. Wildcards did not significantly affect the results of the branched tests, but the non-branched tests did not terminate in reasonable time. The other tests that are omitted are the ones with separate child and descendant steps. These tests gave similar results as the ones with both child and descendant steps, and did not provide any new information. The graphs for the omitted tests can be found in appendix A.

Figure 4.7: Execution times for test 3. Child descendant
Figure 4.8: Execution times for test 9. *Child descendant branch (depth 2)*

![Execution times for test 9](image1)

Figure 4.9: Execution times for test 15. *Child descendant branch (rank 2)*

![Execution times for test 15](image2)
Chapter 5

Discussion

In this chapter, we discuss and analyse the results, the methods and the literature used in this thesis.

5.1 The Method

The method involves implementing and comparing the runtime of algorithms. The most practical way to measure and compare the speed of algorithms is to use benchmarks with realistic sample data. Comparing the time complexity of the algorithms alone does not necessarily translate into practical speed, especially for small sample data like database queries. A flaw in this method is that the implementations might be suboptimal and not perform at their best. This means that the comparison between the algorithms could be different with other implementations. In this thesis, implementations of the two Query Containment algorithms have been compared, rather than the algorithms in general.

The method also involves comparing how easy algorithms are to implement. There is no definitive way of doing this, and the results may vary from implementer to implementer. The results of the questionnaires should therefore be considered an indication to a potential future implementer rather than a scientific study on the ease of implementation of the algorithm. Such a study is not in the scope of this thesis. The results of this criterion are still included since they reflect the authors’ experience implementing the algorithms, and this is still considered relevant for the purpose of this thesis.
The accuracy of the algorithms were compared using a literature study using previous research. The accuracy of the algorithms is an important aspect to consider when evaluating the algorithms. A literature study was chosen for this criterion since this has previously been well researched, e.g. in [10]. One aspect that was not investigated was measuring the rate of false negatives for *The Homomorphism Technique*. This was not considered necessary since a false negative did not affect the overall performance.

Testing the algorithms in a database system can be done in numerous ways. In this thesis a simple database system was implemented with a reduced functionality, keeping only the functionality necessary for the purpose of this thesis. This criterion is considered the most important one, since this thesis investigates practical aspects of Query Containment, and database systems are where the implementations eventually will be useful. The database system that was implemented might not be similar to database systems used in production. A simplified implementation was necessary because of the high complexity of modifying commercial database systems.

There are three aspects of the database system implementation that can affect the results. The first is the querying process. The implementation is naive and could be severely optimised. This aspect was the slowest part of the database system. The second aspect is the algorithms, that could be optimised as well. The third aspect is the process to fetch the results from the previous query’s results. This cannot be done by simply running the query on the previous results, but needs to be done in relation to the database. This implementation is also naive and could be optimised.
5.2. THE RESULTS

5.2 The Results

In this section, the results of the benchmark tests and the implementation questionnaire are discussed.

5.2.1 Speed

Both The Canonical Model and The Homomorphism Technique are polynomial in time when comparing queries with just child step chains, as can be seen in figure 4.1. When comparing descendant step chains, The Canonical Model is exponential while The Homomorphism Technique continues to be polynomial, as can be seen in figure 4.2. If a query $p$ has $d(p)$ descendant steps and the length of the longest child-wildcard chain in a query $q$ is $m(q)$, The Canonical Model has to create $(m(q) + 2)^d(p)$ transformed trees to match against the other query. The Homomorphism Technique can instead directly match the other query against the first, making it faster than The Canonical Model.

When the queries have branching predicates, the matching process grows slower because of the number of possibilities that have to be considered. Since the number of trees that have to be matched in The Canonical Model is exponentially larger than in The Homomorphism Technique, The Canonical Model grows slow quicker than The Homomorphism Technique.

5.2.2 Ease Of Implementation

The Homomorphism Technique was easier to implement than The Canonical Model. This might be because The Homomorphism Technique can be considered a special case of The Canonical Model, and less has to be done in the algorithm. Even though The Homomorphism Technique was easier to implement than The Canonical Model, it was also observed to be better in most aspects, which conflicts with the initial speculation in section 1.2.1.2.

5.2.3 Accuracy

The accuracy of the algorithms were compared with a literature study using previous research. It was concluded that The Canonical Model was accurate in its subset of XPath, thus always giving a correct answer. The Homomorphism Technique on the other hand, can give false negative results for XPath(/, //, [], *). This is however not a major concern in a database system using Query Containment optimisation, since the results will be fetched directly from the database instead and the result of the query will still be correct.
5.2.4 Usability In Database Systems

The two algorithms’ performance did not differ significantly in the database system for reasonably sized queries. The main reason is that the querying process and fetching the results from a contained query are the most time consuming aspects of the system. Which algorithm that was used does not significantly affect the time, but the presence of an algorithm at all means that if there was containment, the results of the second query must be fetched from the results from the first query. This adds overhead to the process and generally slows it down. For child step and descendant step chains, the results were better and actually increased the performance of the system.

These results indicate that Query Containment algorithms might not generally improve the performance of a database system. This could be different with better implementations of the algorithms and the results fetching process. As can be seen in the results in figure 4.7, there is performance to be gained when specific query types are used.

A way that the implementations of this thesis could be used in a commercial database system for optimisation is to only use the algorithms for the XPath subsets that gained performance using the algorithms. According to the results in section 4.4, slightly longer chains of child and descendant steps with no wildcards or branching predicates were the only queries that gained performance using the algorithms. In practice, a database system would have to evaluate on beforehand if the current queries are suitable for a Query Containment optimisation. This process could however add more overhead to the querying process, so this process would have to be investigated and evaluated further. This is however not in the scope of this thesis.

5.3 The Literature

Query Containment for XML Queries is a relatively new field in research, and there are only a few researchers that have published theses in this field. Most of the publications in this field focus on the theoretical aspects of Query Containment. To the authors’ best knowledge, there are no publications on the practical aspects of implementing the algorithms. This means that there is still a huge potential for further research in this field.
Chapter 6

Conclusion

In this chapter, the overall conclusions that have been made for this thesis is presented to the reader, as well as recommendation for future work regarding the subject of Query Containment implementation.

6.1 Conclusions

- *The Homomorphism Technique* is faster than *The Canonical Model* in every test case of this thesis.

- Both *The Homomorphism Technique* and *The Canonical Model* are easy to implement according to the authors, but *The Homomorphism Technique* is slightly easier to implement.

- *The Canonical Model* is more accurate for *XPath(//, //, [], *)* since *The Homomorphism Technique* can give false negatives for this subset of XPath.

- There is performance to be gained using a Query Containment algorithm for *XPath(/), XPath(/ //) and XPath(/, //)* in the database system developed in this thesis.
• For the implementations of the algorithms in this thesis, there is no performance to be gained for:
  
  - $XPath(/, *)$
  - $XPath(//, *)$
  - $XPath(/, //, *)$
  - $XPath(/, [])$
  - $XPath(//, [])$
  - $XPath(/, //, [])$
  - $XPath(/, [], *)$
  - $XPath(//, [], *)$
  - $XPath(/, //, [], *)$

• A database system utilising the Query Containment algorithms in this thesis should for every issued query evaluate if a Query Containment optimisation should be used.

6.2 Recommendations

The authors’ recommendations regarding Query Containment algorithms are that The Homomorphism Technique should always be used rather than The Canonical Model, even though the latter is more accurate. The accuracy of The Canonical Model comes at the cost of speed, and makes the algorithm too slow to be used in practical systems.

In a database system, using a Query Containment optimisation is in most cases not beneficial. If a Query Containment algorithm is to be used in a database system, every query should be evaluated on beforehand when parsing the query to determine if the algorithm should be used for optimising the querying process.
6.3. Future work

There are more Query Containment algorithms to be tested, e.g. *The Automata Technique* and *The Chase Technique* [10]. These algorithms might be more efficient than the ones tested in this thesis.

Approximation algorithms and heuristics for the Query Containment problem might also be more efficient than the algorithms used in this thesis. An evaluation of the rate of false results would have to be made in that case.

Both the algorithms and the database system in this thesis could be optimised to gain better performance, and hence gain more support for using Query Containment optimisation in a database system.

The implementations of the algorithms in this thesis could be implemented as a query optimisation in a complete database system. The performance gain would have to be measured in detail and evaluated.


Appendix A

Code

The code and implementations, as well as the graphs produced in this thesis can be found on GitHub under the address

    github.com/WareusWallstedt/QueryContainmentXPathSystem.

This repository contains all the implementations of the algorithms and the benchmarks tests described in this thesis. The algorithms were implemented in the language C, and most of the benchmark tests were developed in Bash scripts and MATLAB, using GNU Octave.
Appendix B

Grammars

B.1 XML

```
document ::= prolog element s*
prolog ::= ('<?xml' s 'version' s? '=' s? ((""", '"1.', [0-9]* ",") | (""", '"1.' [0-9]* ",")) [^?]* '?>')?

s?

(element ::= EmptyElemTag | STag content ETag)

EmptyElemTag ::= '<' name [^/>] ' />'

STag ::= '<' name [^>] '*' ' >'

ETag ::= '</' name s? '>'

collection ::= CharData? (element CharData?)*

CharData ::= [^<]*
s ::= (#x20 | #x9 | #xD | #xA)+
name ::= (':', | [A-Z] | '_', | [a-z])

([':', | [A-Z] | '_', | [a-z] | ^'-' | [0-9] | '~',]*

ExtID ::= 'SYSTEM' s ( (""", '['',]""",')['","]",.Concat(""",",")")
```

Listing B.1: The grammar used in the parser for XML.

B.2 XPath

```
XPath ::= expr
expr ::= ('/' | '//')? (name | '*') pred* step?
name ::= (':', | [A-Z] | '_' | [a-z])

((':', | [A-Z] | '_' | [a-z] | ^'-' | [0-9] | '~',)*

pred ::= ('[' expr ']')

step ::= ('/' | '//') (name | '*') pred* step?
```

Listing B.2: The grammar used in the parser for XPath.
Appendix C

Ease of Implementation Questionnaire

C.1 Template

In this questionnaire, the leftmost answer is worth 1 point, and the rightmost is worth 5 points. A mean score is calculated for every algorithm implemented. A lower mean score means that the algorithm was easier to implement and a higher mean score means that the algorithm was harder to implement.

1. Approximately how long did it take to implement the algorithm?

<table>
<thead>
<tr>
<th>0h-4h</th>
<th>4h-12h</th>
<th>12h-20h</th>
<th>20h-40h</th>
<th>40h+</th>
</tr>
</thead>
</table>

2. How difficult was the algorithm to understand?

<table>
<thead>
<tr>
<th>Very easy</th>
<th>Easy</th>
<th>Normal</th>
<th>Hard</th>
<th>Very hard</th>
</tr>
</thead>
</table>

3. How difficult was the algorithm to implement?

<table>
<thead>
<tr>
<th>Very easy</th>
<th>Easy</th>
<th>Normal</th>
<th>Hard</th>
<th>Very hard</th>
</tr>
</thead>
</table>

4. How satisfied are you with the results?

<table>
<thead>
<tr>
<th>Very satisfied</th>
<th>Satisfied</th>
<th>Indifferent</th>
<th>Dissatisfied</th>
<th>Very dissatisfied</th>
</tr>
</thead>
</table>

Total score:   Mean score:
## C.2 The Canonical Model

1. Approximately how long did it take to implement the algorithm?

<table>
<thead>
<tr>
<th>Time Frame</th>
<th>0h-4h</th>
<th>4h-12h</th>
<th>12h-20h</th>
<th>20h-40h</th>
<th>40h+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choice</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

2. How difficult was the algorithm to understand?

<table>
<thead>
<tr>
<th>Difficulty Level</th>
<th>Very easy</th>
<th>Easy</th>
<th>Normal</th>
<th>Hard</th>
<th>Very hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choice</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

3. How difficult was the algorithm to implement?

<table>
<thead>
<tr>
<th>Difficulty Level</th>
<th>Very easy</th>
<th>Easy</th>
<th>Normal</th>
<th>Hard</th>
<th>Very hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choice</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

4. How satisfied are you with the results?

<table>
<thead>
<tr>
<th>Satisfaction Level</th>
<th>Very satisfied</th>
<th>Satisfied</th>
<th>Indifferent</th>
<th>Dissatisfied</th>
<th>Very dissatisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choice</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Total score: 10  
Mean score: 2.5
C.3. THE HOMOMORPHISM TECHNIQUE

C.3 The Homomorphism Technique

1. Approximately how long did it take to implement the algorithm?

<table>
<thead>
<tr>
<th></th>
<th>0h-4h</th>
<th>4h-12h</th>
<th>12h-20h</th>
<th>20h-40h</th>
<th>40h+</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. How difficult was the algorithm to understand?

<table>
<thead>
<tr>
<th></th>
<th>Very easy</th>
<th>Easy</th>
<th>Normal</th>
<th>Hard</th>
<th>Very hard</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. How difficult was the algorithm to implement?

<table>
<thead>
<tr>
<th></th>
<th>Very easy</th>
<th>Easy</th>
<th>Normal</th>
<th>Hard</th>
<th>Very hard</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4. How satisfied are you with the results?

<table>
<thead>
<tr>
<th></th>
<th>Very satisfied</th>
<th>Satisfied</th>
<th>Indifferent</th>
<th>Dissatisfied</th>
<th>Very dissatisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total score: 5
Mean score: 1.25