TOWARDS RELEVANCE BASED VIEWS IN XML QUERY LANGUAGES

Sarita Divakaran
ABSTRACT

TOWARDS RELEVANCE BASED VIEWS IN XML QUERY LANGUAGES

Numerous XML query languages have been proposed and developed in the recent past to retrieve information from structured and unstructured documents. While these query languages resolve many challenges incurred with the increased use of XML for information processing and storage, there is little emphasis on the result-view provided to the query user for document oriented collection. To overcome this shortcoming, Junkkari et al. [2006] proposed a new query language called XIL (XML information retrieval Language). XIL reconstructs the results for document intensive collections and provides a relevance based view to the query user.

This thesis checks the feasibility of constructing relevance based views using Dewey indices for XIL and further investigates on its design and development. The detailed syntax for XIL is also specified, in EBNF(Extended Backus–Naur Form) notation, as part of this study.

As background research, four existing languages (XPath, XIRQL, XQuery and TeXQuery) were studied for its syntax, features, data model and results shown to the end user. The study concludes by investigating if relevance based views offered by XIL is beneficial from an end user point of view in comparison to these four query languages apart from validating the benefits outlined in XIL.

Key words and terms: XIL, XML Query Language, XML Search, Relevance Ranking
ACKNOWLEDGEMENTS

I would like to take this opportunity to thank my supervisors Dr. Marko Junkkari and Dr. Jaana Kekäläinen for the support they have provided me during this thesis work and my studies. I am grateful for the opportunity provided by them for my summer internship at the university. It was extremely rewarding to understand the challenges of developing a new query language and observe the way of working in academic circle first hand. It truly has been a pleasure being under their tutelage.

I would also like to extend my thanks to staff members of both the Computer Science and Information Science department for making my study a wonderful experience. A special thanks to Mr. Paavo Arvola for his the tips and suggestion provided during my internship.

My sincere thanks to Ms. Kirsi-Marja Tuominen for always having her door open, providing a patient ear and giving her views and support throughout my masters program.

At a personal level I'd like to thanks my parents, sisters, in-laws, family and friends for their love and encouragement I have received over the years. Last but definitely not least I'm extremely grateful to my husband, Lal, for being supportive all through my studies; this journey would have been difficult without it.
GLOSSARY

API  Application Programming Interface
CAS  Content and Structure
CO   Content Only
EBNF Extended Backus–Naur Form
DNF  Disjunctive Normal Form
DOM  Document Object Model
DTD  Document Type Definition
FLWOR FOR, LET, WHERE, ORDER BY, RETURN
HTML Hyper Text Markup Language
IR   Information Retrieval
LAMP Linux, Apache, MySQL, PHP
OQL  Object Query Language
PHP  Hypertext Preprocessor
SGML Standard Generalized Markup Language
SQL  Structured Query Language
TRIX Tampere retrieval and indexing system for XML
URI  Uniform Resource Identifier
URL  Uniform Resource Locator
W3C World Wide Web Consortium
XIL  XML Information retrieval Language
XIRQL XML Information Retrieval Query Language
XML  Extensible Mark-Up Language
XQL  XML Query Language
# Table of Contents

ABSTRACT .....................................................................................................ii
LIST OF TABLES .........................................................................................vii
LIST OF FIGURES .......................................................................................viii
LIST OF FIGURES .......................................................................................viii

Chapter 1. INTRODUCTION ...........................................................................1
  1.1 General Background ...........................................................................1
  1.2 Aims and Research Questions ..............................................................1
  1.3 Methods ..............................................................................................2
  1.4 Thesis Outline ...................................................................................3

Chapter 2. XML AND XML QUERY LANGUAGES .......................................4
  2.1 Introduction to XML ...........................................................................4
  2.2 XML Query Languages .......................................................................5
    2.2.1 Query Languages Based on SQL or OQL ......................................5
    2.2.2 Query Languages Based on Path Expressions ..............................6
    2.2.3 Query Languages Based on Views ..............................................6
    2.2.4 Summary of XML Query Languages ...........................................8
  2.3 XML Path Language ...........................................................................8
    2.3.1 Overview .....................................................................................8
    2.3.2 Features .....................................................................................9
    2.3.3 Syntax .......................................................................................9
    2.3.4 Data Model ...............................................................................10
  2.4 XIRQL ..................................................................................................11
    2.4.1 Overview .....................................................................................11
    2.4.2 Features .....................................................................................11
    2.4.3 Syntax .......................................................................................12
    2.4.4 Data Model ...............................................................................12
  2.5 XQuery ..................................................................................................13
    2.5.1 Overview .....................................................................................13
    2.5.2 Features .....................................................................................13
    2.5.3 Syntax .......................................................................................14
    2.5.4 Data Model ...............................................................................15
  2.6 TeXQuery ................................................................................................15
    2.6.1 Overview .....................................................................................15
    2.6.2 Features .....................................................................................15
    2.6.3 Syntax .......................................................................................16
    2.6.4 Data Model ...............................................................................16
  2.7 XIL .......................................................................................................18
    2.7.1 Overview .....................................................................................18
    2.7.2 Features .....................................................................................18
    2.7.3 Syntax .......................................................................................18
    2.7.4 Data Model ...............................................................................19
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>DTD used for the sample XML documents used in the queries</td>
<td>4</td>
</tr>
<tr>
<td>Table 2</td>
<td>FullMatch DTD for TeXQuery</td>
<td>17</td>
</tr>
<tr>
<td>Table 3</td>
<td>Example of Dewey Ordering</td>
<td>20</td>
</tr>
<tr>
<td>Table 4</td>
<td>Proposed XIL syntax in EBNF notation</td>
<td>21</td>
</tr>
<tr>
<td>Table 5</td>
<td>Application Environment</td>
<td>30</td>
</tr>
<tr>
<td>Table 6</td>
<td>Structure of the q-tree for XIL</td>
<td>35</td>
</tr>
<tr>
<td>Table 7</td>
<td>Example for constructing the d-tree returned from TRIX</td>
<td>36</td>
</tr>
<tr>
<td>Table 8</td>
<td>Structure of d-tree for sample XML</td>
<td>37</td>
</tr>
<tr>
<td>Table 9</td>
<td>XPath queries and results</td>
<td>41</td>
</tr>
<tr>
<td>Table 10</td>
<td>XIRQL queries and results</td>
<td>44</td>
</tr>
<tr>
<td>Table 11</td>
<td>XQuery queries and results</td>
<td>46</td>
</tr>
<tr>
<td>Table 12</td>
<td>TeXQuery queries and results</td>
<td>49</td>
</tr>
<tr>
<td>Table 13</td>
<td>Comparing query languages for query formulation</td>
<td>52</td>
</tr>
<tr>
<td>Table 14</td>
<td>Comparing query languages for handling CAS and CO queries</td>
<td>52</td>
</tr>
<tr>
<td>Table 15</td>
<td>Comparing query languages for handling heterogeneous collection</td>
<td>53</td>
</tr>
<tr>
<td>Table 16</td>
<td>Comparing query languages for relevance based views</td>
<td>53</td>
</tr>
<tr>
<td>Table 17</td>
<td>Summary of comparison of XML query languages</td>
<td>54</td>
</tr>
<tr>
<td>Table 18</td>
<td>Sample XML file, article.xml</td>
<td>62</td>
</tr>
<tr>
<td>Table 19</td>
<td>Results for TexQuery for a search on logic program</td>
<td>71</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Subdivision of XML Query Languages</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2</td>
<td>XIL High-level Architecture</td>
<td>28</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Sequence diagram for a XIL query with about clause</td>
<td>29</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Query Processing Flow across Interfaces</td>
<td>31</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Simple UI for XIL Query</td>
<td>32</td>
</tr>
<tr>
<td>Figure 6</td>
<td>XIL Query UI with Multiple Query fields example</td>
<td>33</td>
</tr>
</tbody>
</table>
Chapter 1. INTRODUCTION

1.1 General Background

XML, Extensible Markup Language, was first developed by a Working Group formed under the auspices of the World Wide Web Consortium (W3C) in 1996 [W3C XML 1.0, 2006]. It is a meta-data language that describes the data in a simple manner and separates the data from its formatting rules. XML facilitates information interchange by allowing different computer systems to work together through the exchange of everything from simple numbers to elaborate data structures and human-readable texts [Goldfarb and Prescod, 2004].

The various features offered by XML have led to its emergence as a preferred standard for transferring information especially over the Internet. Enormous amounts of data such as electronic journals and libraries, health care information, learning, news, e-business, multimedia files (such as video, image, sound) are stored in XML format [Baydaa et al., 2008]. This large volume of data stored in XML requires tools to carry out search operations and efficiently retrieve information. This has propelled the need for an XML query language. A number of them have been developed over the last decade to address various issues that are faced in retrieving relevant parts of the XML document. Some XML query languages that have been developed take advantage of the intuitive XML structure and are simple to use; while others have incorporated sophisticated Information Retrieval (IR) features such as weighting, ranking, and relevance oriented search. The end user is thus able to choose query languages that are better suited to their needs and ability.

1.2 Aims and Research Questions

The purpose of this thesis is to research and analyze further the proposal for a relevance based XML query language called XML Information Retrieval Language (XIL, \( \xi \ell \)). XIL is a query language for document-oriented XML retrieval which also has characteristics of data-
oriented query language [Junkkari et al., 2006]. The semantics of XIL is unique as it exploits the capabilities of Dewey-like structural indices\(^1\) while constructing the result for the input query. XIL proposes to present the retrieved result to be composed of views of relevant elements distinctly for each document.

The research investigates the various benefits of XIL by implementing some of the basic queries in the proposal. It further verifies the benefits of XIL outlined by Junkkari et al. [2006]. This requires a detailed study on some of the existing XML query languages widely in use. This research classifies these languages into categories and chooses a sample representative for each category. This categorization has a dual purpose, the first is to subdivide the query languages into distinct groups; the second is to simplify the study of a particular language within a category.

The challenges faced when developing a user friendly query language range from being able to formulate queries using simple syntax and ensuring that the most relevant information is retrieved to presenting the result to the user in a manner that allows them to process the information better. This thesis investigates the presentation of query results for various selected query languages while focusing on XIL.

1.3 Methods

First, this thesis investigated the existing XML query languages. Most of these query languages are based on SQL, Structured Query Language, or path expression (or XML structure). These categories of query languages can be further sub-divided into two: data-centric or document-centric. A survey was carried out to identify these aspects of the available query languages.

Four query languages fulfilling the data-centric or document-centric criteria are chosen for further investigation. The salient features of each of these chosen query languages are

\(^1\) Dewey indices are covered in detail chapter 2
studied; mainly their syntax and the structure of their query result. During this investigation, any benefits or problems encountered in these existing languages were noted and were used as pointers while analyzing XIL.

The next stage investigated the features proposed in XIL. It checked the feasibility of constructing relevance based views using Dewey indices. Sample queries were processed based on the rules defined in language and this provided the input for the implementation proposal. The implementation proposal also delves into matters such as user friendly interface, ease in query formulation and different possibilities of presenting the view to the user.

Finally, the study concludes by investigating if relevance based views offered by XIL is beneficial from an end user point of view in comparison to the selected query languages.

1.4 Thesis Outline

This thesis is comprised of six chapters. The introduction chapter to this thesis provides the motives for this study and its aims. Chapter 2 presents the relevant background literature for the thesis project and briefly looks into XML and XML query languages. It also further investigates the four XML query languages selected for this study and introduces XIL to the readers. Chapter 3 is devoted entirely to XIL, elaborating on its features and data model.

Chapter 4 then looks into the implementation proposal of XIL, the system overview with an algorithm for the main functionality. Chapter 5 provides examples for the investigated query languages and compares the view of the result set.

Chapter 6 concludes this research and looks at the prospects for future investigation. References and appendices are provided as pointers to the readers.
Chapter 2. XML AND XML QUERY LANGUAGES

2.1 Introduction to XML

XML is a simplified subset of Standard Generalized Markup Language (SGML) similar to HTML. XML and HTML documents conform to SGML rules. As SGML was too complex to be used for the Web, HTML was developed to create Web accessible documents. HTML was specifically designed to format data for Web browsers but uses only a fixed set of element types (tag names). XML in contrast, follows the fundamental concept of separating the logical structure of a document from its layout. The major purpose of XML markup is the explicit representation of the logical structure of a document within markup tags [Fuhr and Großjohann, 2001]. This ensures that XML documents are self-describing as the markup tags surrounds the data in the document and provides a context to it.

Due to the logical markup of XML documents, different kinds of operations referring to the logical structure can be performed. XML supports interoperability by making it simple to exchange information in the logical structure between different software systems. Hence, XML is acknowledged as a standard document format, especially for Web applications.

XML documents use DTDs, Document Type Definitions, to ensure that the context of the document is valid against a set of rules. This ensures that an XML document conforms to the expectations of an application before it is processed. For this study, the XML collection consists of articles from scientific journals and adheres to the DTD given in Table 1 below:

Table 1 DTD used for the sample XML documents used in the queries

```
<! ELEMENT article (title, author+, abstract, index_term?, sec+, reference*, author_biography*)>
<! ELEMENT sec (title, p*, sub_sec*)>
<! ELEMENT sub_sec (title, p*)>
```
2.2 XML Query Languages

The increased use of data storage in XML format has given rise to numerous query languages being developed. Chapter 2.2.3 classifies XML query languages based on views. Another approach is to categorize based on traditional database query languages, i.e. SQL [Chamberlin and Boyce, 1974] or OQL (Object Query Language) [Cluet, 1998], and those which are inspired by XML structure, i.e. based purely on path expression. In the following sections we look into these sub-divisions in detail.

2.2.1 Query Languages Based on SQL or OQL

The database community work has established SQL as a standard for retrieving data from relational databases and OQL as a standard for object-oriented databases. Database query languages developed were expressive yet simple and very powerful [Fernandez et al., 1999]. Hence most of the early works of storing, indexing, querying, and searching XML documents were based on the database communities work on semi-structured data such as Lorel [Fernandez et al., 1999] and XML-QL [Deutsch et al., 1998].

An extension of SQL, SQL/XML [Eisenberg and Melton, 2001; 2004], has been developed to exploit completely the capabilities offered by SQL. However, XML data and relational data differ in some major aspects. This has lead to the development of languages such as XQuery which are only loosely based on SQL. For example, the from-let-where-order-return (FLWOR) expression is similar to the select-from-where SQL query expression in the sense
that both are block-structured query languages. Also XQuery has various grouping and joins which are based on SQL. [Chamberlin, 2004]

2.2.2 Query Languages Based on Path Expressions

An XML document is also represented as a tree hierarchy with the root on top and leaves at the bottom. All the entities in the XML document are classified as nodes in the tree depending on their properties. The first element in the XML document is represented as the root node and it does not contain any attributes or text content. Similarly, document entity, elements, attributes, processing instructions, and comments are also represented in the tree structure as nodes [Robie et al., 1999]. This representation of the XML document, based on its structure, contains information that can be exploited by query languages. The tree structure maintains parent-child and ancestor-descendant hierarchical relations and a position (absolute or relative) of elements within sibling list. These features have been extended and used in various XML query languages such as XPath [W3C XPath 2.0, 2007], XQL [Robie et al., 1999] and NEXI [Trotman and Sigurbjörnsson, 2005].

2.2.3 Query Languages Based on Views

XML applications can be classified into data-centric and document-centric. Data-centric applications use XML for exchanging formatted data in a generic, serialized form between different applications. On the other hand, document-centric applications use XML as a format for representing the logical structure of documents. Fuhr and Großjohann [2004] categorize XML query languages in a similar manner based on the query results as given below:

1) The data-centric approach requires a query language that allows for selection as well as restructuring (of result documents) and aggregation operators (e.g. count, sum).
2) The document-centric approach requires a query language that mainly supports selection based on conditions with respect to both structure and content, taking into account the intrinsic uncertainty and vagueness of content-based retrieval.

The data-centric approach has certain clear advantages. For example, the data relations, constraints and integrity can be modeled and checked. Standard database can be used to store and retrieve the data. Besides this, the query language used is similar to the well known standard database query language like SQL or OQL. On the other hand, importing XML data into a database in data-centric approach is difficult and is relatively hard to handle schema changes. XQuery is a query language which focuses on data-centric views. It is more database-oriented and provides almost no support for the IR-oriented querying of XML documents. According to the definition in W3C XQuery 1.0 [2007] “XQuery operates on the abstract, logical structure of an XML document, rather than its surface syntax. This logical structure is known as the data model.”

Until the turn of this century, very little attention was given to document-centric approach where the XML documents are viewed as a collection of text documents with additional tags and relations between these tags. Traditionally, IR techniques had been applied to search large sets of textual data and this was extended to encode the structure and semantics inherent in XML document. Integrating powerful IR techniques such as document similarity ranking and keyword search to XML search enabled sophisticated search to be carried out on the structure as well as the content of these documents [Carmel et al., 2001]. Unlike systems that address the data-centric issues of XML, content-oriented XML retrieval systems take a document-centric view on XML and exploit the explicitly represented logical structure of documents in order to retrieve document components (instead of whole documents) in response to a user query [Kazai et al., 2004].

Document-centric user queries follow two approaches discussed below:
1) Content only (CO): The query specifies only content-oriented conditions and no restrictions to specific structural elements are made.

2) Content and structure (CAS): The query condition is specified with respect to the structure of documents and type of result elements.

2.2.4 Summary of XML Query Languages

Figure 1 summarizes the discussions above to classify XML query languages.

![Figure 1 Subdivision of XML Query Languages](image)

As more and more XML query languages are developed useful features are borrowed from categories of other languages and it may not be possible to uniquely classify the languages based on Figure 1. For example, TeXQuery is an extension on XQuery so it also uses path expression in query formulation.

2.3 XML Path Language

2.3.1 Overview

XML Path Language or XPath is an expression language for addressing parts of an XML document and for computing values (strings, numbers, or Boolean values) based on the
content of an XML document [W3C XPath 2.0, 2007]. XPath gets its name from the use of a path notation similar to computing file system expression or URIs, for navigating through the hierarchical structure of an XML document. It models an XML document as a tree of nodes and provides the ability to navigate around the tree by selecting nodes based on various criteria thereby identifying particular parts easily in XML documents.

2.3.2 Features

XPath uses path expressions to select nodes and navigate through a single source XML document. It can be written intuitively and read using familiar characters and constructs. The XPath expression consists of sequence of steps and zero or more predicates.

A step is separated by a ‘/’, to get from one XML node to another node or a set of nodes. It may either be an “axis step” or a “filter expression” [W3C XPath 2.0, 2007]. An “axis step” returns the relationship as to where are you going with respect to the evaluation context (e.g. child:: or parent::). The evaluation context is otherwise called the “node test” and could be based on the kind of node (e.g. element), the name of the node (element or attribute name in the XML) or the type of node (schema defined type).

The step could have predicates which are any kind of expressions used to further refine the set of nodes selected by the step. Predicates are always embedded in square brackets.

2.3.3 Syntax

For the sake of simplicity, only abbreviated syntax in W3C XPath 2.0 [2007] is considered; a few as cited below:

- `/xyz` returns the root element with name `xyz`
- `//xyz` returns the element `xyz` in the document from the context menu
- `@xyz` returns attributes that have the name `xyz`
• selects all elements in preceding path, hence */ selects the entire document

• /article/sec[5]/sub_sec[2] selects the second sub-section of the fifth section of the article

• /article[author='Anonymous']/title selects all title in the article with author as ‘Anonymous’; the condition is specified within the square brackets [].

2.3.4 Data Model

In XPath, every instance of the data model is a sequence which is an ordered collection of zero or more items [W3C XDM, 2007]. All items are either atomic values (e.g. strings, Boolean, decimals integers, floats and doubles, and dates) or a node of one of the seven types below:

• Document nodes: represents top of the hierarchy in an XML document and encapsulate the entire XML document itself

• Element nodes: represents any element in an XML document

• Attribute nodes: represents those elements that have attribute values

• Namespace nodes: represents those elements that have an associated set of namespace nodes

• Processing instruction nodes: encapsulate XML processing instructions

• Comment nodes: encapsulates the comments in XML documents delimited by <!-- and --!>

• Text nodes: encapsulate the textual data content within an element in XML other than data in attribute value, processing instructions and comments.
2.4 XIRQL

2.4.1 Overview

XIRQL ("circle") [Fuhr and Großjohann, 2001] is a query language that combines the major concepts of XML querying with those from IR. It is based on XQL, which retrieves the sub-trees of the XML document fulfilling the specified condition (e.g. child operator "/", descendent operator "//", filter operator, support for dates as data types, sub queries combined by Boolean operators). XQL allows flexible formulation of conditions with respect to structure and content of XML documents and the result is always a set of elements from the original document(s); hence following the document-centric view. XIRQL has extended features to that of XQL, as described in the following chapter.

2.4.2 Features

The core features of XIRQL are as follows [Fuhr and Großjohann, 2001]:

1) Weighting: For effective retrieval in textual documents it has been proven that document term weighting as well as query term weighting are necessary tools. This is further used for document ranking.

2) Specificity-oriented Search: Only the requested content is specified, but not the type of elements to be retrieved. Hence the most specific elements that match the query must be retrieved as the relevant elements.

3) Data types with vague predicates: The standard IR approach supports vague searches on plain text must be extended for XML documents. As XML allows for a fine grained markup of elements there should be the possibility to use special search predicates for different types of elements e.g. names should have similarity search for proper names, age should have similarity checks as teenage is equivalent to 13-19.
4) Structural vagueness: This is achieved by dropping distinction between attribute and element or by using an ontology over element name e.g. country-continent.

2.4.3 Syntax

XIRQL is an extension of XQL [Robie et al., 1999] and hence a typical XQL syntax is considered:

```
//article[@author = <search pattern #01> $and$ chapter/heading = <search pattern #02>] searches for articles authored by <search pattern #01> which have a chapter about <search pattern #02>.
```

XIRQL extends XQL syntax by introducing $cw$, contains word, to test the word occurrence in an element and $clsim$ to test class similarity.

```
//section[heading $cw$ "syntax"] searches for sections containing “syntax” in heading.

//article[@id clsim “1.2.3”] returns articles with attribute named id similar to “1.2.3”
```

To support features like structural vagueness, XIRQL introduces the following operators

```
//=author returns element and attribute that have name author
sec\title returns title elements in section and sub-sections but weights the title element in section higher than those in sub-section.
```

2.4.4 Data Model

To assign weights to an XML document it is split into atomic units that are disjoint with possibility to retrieve specific parts. In XIRQL, nodes of a specific type are pre-selected and set as atomic units based on the concept of “index nodes” from FERMI multimedia model [Fuhr and Großjohann, 2004]. In XIRQL queries, the occurrence of a search term in an index node is done using the comparison predicate $cw$ (contains word). This is known as a basic event and is given as [node id, term]. Here, either the terms vary or the same
term is searched in the different index nodes thereby ensuring that the basic events occur independently. Based on the conditions specified in the query, the basic events in a document are combined using Boolean expressions, like $\land \lor \neg$. This resulting combination that describes the relationship between the answer element and the query is called as Event Expressions [Fuhr and Großjohann, 2004]. Event Expressions are evaluated by transforming them as disjunctive normal form (DNF) and applying the inclusion-exclusion formula to calculate their probability.

2.5 XQuery

2.5.1 Overview

XQuery is a powerful language for querying over XML data built on XPath expression. It is a W3C recommendation and is supported by all the major database engines such as IBM, Oracle and Microsoft.

2.5.2 Features

The key features offered by XQuery are:

1) Input functions: There are two input functions to identify the data to be queried.

   Function `doc()` retrieves the document node and function `collection()` returns the collection.

2) Path Expression: The path expressions are extended from XPath to locate the nodes from a XML document. These expressions are similar to that mentioned in the XPath section.

3) Creating Nodes: Nodes can be created in XQuery by defining the query between curly braces `{ }`. The terms enclosed in the braces are evaluated to create element, attribute, text, comment or processing instruction nodes as in XML.
4) Combining and restructuring nodes: (FLWOR expressions): FLWOR is an acronym for 'For, Let, Where, Order by, Return. This is a powerful expression that is similar to the SELECT-FROM-WHERE in SQL. Unlike SQL which is defined in terms of tables, rows and columns FLWOR expression binds variables to form tuples in for and let clause and uses these tuples to create new results [Robie J, 2003]. The where clause filters the tuples based on the condition specified, order by clause sorts the tuples and the return clause builds the result for the FLWOR expression.

5) Conditional Expression/ Arithmetic operators: XQuery uses IF-THEN-ELSE construct in conditional expression as in other languages. To evaluate the condition it also supports arithmetic (+, -, *, div, idiv, and mod) and comparison operators. The comparison operator include less than (<), greater than (>), equals (=), less than or equal to (<=), greater than or equal to (>=), and not equal to (!=).

2.5.3 Syntax

FLOWR expression syntax can be given by the following expression [W3C XQuery 1.0, 2007]:

\[
\text{FLOWR Query ::= (ForClause | LetClause)+} \\
\text{\hspace{1cm} (“where” Expression)} \\
\text{\hspace{1cm} (OrderByClause)} \\
\text{\hspace{1cm} “return” Expression} \\
\text{ForClause ::= “for” $VariableName (“at” $PositionalVariableName) “in” Expression} \\
\text{LetClause ::= “let” $VariableName “:=” Expression} \\
\text{OrderByClause ::= “order by” Expression (“ascending”|”descending”)}
\]

VariableName is the name of the variable to bind to the result of Expression.
PositionalVariableName is the name of an optional variable that is bound to the position within the input stream of the item that is bound by each iteration of the for clause.

Expression is any XQuery expression. If the expression includes a top-level comma operator, then the expression must be enclosed in parentheses.

2.5.4 Data Model

XQuery / XPath Data Model (XDM) are the same as mentioned in XPath. “XQuery is defined in terms of the XQuery 1.0 and XPath 2.0 Data Model [XQ-DM]², which represents the parsed structure of an XML document as an ordered, labeled tree in which nodes have identity and may be associated with simple or complex types” [Robie J, 2003].

2.6 TeXQuery

2.6.1 Overview

TeXQuery was developed as a predecessor of the “Full Text Search” (FTS) language extension to XPath and XQuery; i.e. W3C XQuery Full-Text [W3C XQuery Full Text, 2008]. It was built to provide an extension for XQuery with very powerful full-text search primitives to allow users to seamlessly query over both structure data and text [Amer-Yahia et al., 2004]. It also provides a flexible scoring construct that can return top-k queries.

2.6.2 Features

The essential features of TeXQuery are as discussed below.

1) It retains the capabilities of XQuery to provide powerful structured queries and extends to carry out full-text search on the content of the XML documents [Botev et al., 2004].

2) To carry out full-text search using TeXQuery a set of powerful full-text primitives (Boolean connectives, phrase matching, proximity distance, stemming, thesauri,

² [XM-DM] is given by [W3C XDM, 2007]
etcetera) that were natural for querying XML are defined. These primitives can be combined with each other to form very complex full-text searches.

3) Relevance ranking is an essential feature for any full-text query languages. Hence, TeXQuery ranks results to support threshold and return top-k queries.

4) TeXQuery data model, “FullMatch”, is mapped to the XQuery Data model. This allows TeXQuery queries to be embedded in XQuery and vice-versa.

2.6.3 Syntax
TeXQuery syntax can be given as follows:

- **Select String:** Expression “fcontains” SearchTerm “with stems”
- **Exclude String:** Expression “fcontains” “!”SearchTerm
- **Select scope:** Expression “fcontains” SearchTerm (“same”|”different”) (”node”|”para”|”sentence”)
- **Window size:** Expression “fcontains” SearchTerm “window 5” (any integer value)
- **Score:** Expression “ftscore” SearchTerm “weight 0.3” (any integer value)

**Expression** is any XQuery expression that is being evaluated.

**SearchTerm** is the name of the variable to bind to the result of Expression.

2.6.4 Data Model
To incorporate the above mentioned features in TeXQuery, a set of powerful full-text search primitives called “FTSelections” is defined [Amer-Yahia et al., 2004]. These can be arbitrarily composed and complex full-text queries can be created by combining the basic
FTSelections. The XQuery data model was inadequate to extend for full-text search primitives. Hence, the “FullMatch” data model was defined. FullMatch data model retains the hierarchical structure of an XML document. In addition, it contains information about the search token (uniquely identifying each token, the node containing the token and, the context) and its relative position of the token in a paragraph or sentence.

Given a TeXQuery query, each search token is expanded to list all search tokens and the position of each search token is determined. For example, ‘with stems’ is evaluated to give all the stemming possibilities. For each position of a search token a SimpleMatch model is created containing information if the search token is included or excluded in the query. The result of this SimpleMatch is nested under FullMatch. In a TeXQuery query, if two search tokens are combined using the Boolean operator && then one SimpleMatch from the first FullMatch is combined with one SimpleMatch of the other FullMatch to give the resultant FullMatch. Thus, a FullMatch model does not provide a list of XML nodes that satisfy the TeXQuery query but rather specifies the positional condition a node should satisfy and is represented by the following DTD [Amer-Yahia et al., 2004] given in Table 2.

**Table 2 FullMatch DTD for TeXQuery**

```xml
<!ELEMENT FullMatch (SimpleMatch)>
<!ELEMENT SimpleMatch (StringInclude| StringExclude)>
<!ELEMENT StringInclude Position>
<!ELEMENT StringExclude Position>
<!ELEMENT Position (Token, Identifier, Node, Sentence, Para, Context)>
```

Thus, the semantics of each FTSelection can be represented as a transformation taking an XML FullMatch input and producing an XML FullMatch output. This XML-to-XML transformation can be specified in XQuery and thus TeXQuery expression can be nested within XQuery/XPath expressions and vice-versa.
2.7 XIL

2.7.1 Overview

XIL, ξℒ, (XML Information Retrieval Language) is a query language whose syntax and semantics developed in University of Tampere, Finland. This query language is based on SQL syntax and is capable of document-oriented XML retrieval. XIL proposes a new method for restructuring the retrieved result-view by exploiting the capabilities of Dewey-like indices.

2.7.2 Features

The major features that are integrated into this IR-oriented query language are [Junkkari et al., 2006]:

1) Declarative query formulation

2) Support for queries based on content and structure or content only

3) Support for querying truly heterogeneous XML collections

4) Views based on relevance ranking.

These points are explained further in Chapter 3.2.

2.7.3 Syntax

XIL is based on SQL-like language and the query has the form

```
select <select_expression>
<from-where_expression>₁
.....
<from-where_expression>ₙ
```
<select_expression> is an expression for specifying the target elements and <fromwhere_expression> determines a branch of the tree associated with the query. As XIL employees the SQL-like syntax, it is able to exploit the declarative, intuitive and powerful nature of SQL.

2.7.4 Data Model

The XIL data model, q-tree, has all relevant information about elements in the collection. It contains the name, index, relevance indicator and weight details of elements. The index parameter is based on Dewey’s order and the following part of this section introduces various ordering methods.

XML has a tree-like structure (or grove-like) and there are various valid numbering schemes which encapsulate this structural order. These ordering encoding methods make it easier to build generic algorithms to automatically traverse the hierarchical tree structure. There are various ordering methods like global, local or Dewey’s order [Tatarinov et al., 2002]. In global order, each node is assigned a number that represents the node’s absolute position in the document. In local order, each node is assigned a number that represents its relative position among its siblings.

Dewey’s order is a hybrid of the above two methods. In this, each node is assigned a vector that represents the path from the document's root to the node and each component of the path represents the local order of an ancestor node [Tatarinov et al., 2002].

Table 3 provides an example of Dewey ordering that is marked in the XML document as an attribute value. The first element collection has not been assigned Dewey index. This was done on purpose to ensure that all article elements have their Dewey indices as single digits as <1>, <2>, <3> ... instead of <1.1>, <1.2>, <1.3>, etcetera.
The notation used in Dewey’s indices makes it possible to easily identify the ancestors and
descendant of any element. For example, the ancestors of element with index <1.4.2.1> are
<1.4.2>, <1.4> and<1>. Also the descendants of <1.4> can be any elements <1.4.x> where
x is non-empty part of the index, in the above example valid descendant of <1.4> are
<1.4.1>, <1.4.2>, <1.4.3> etcetera. XIL makes effective use of these properties offered by
Dewey indices in constructing views based on relevance.
Chapter 3. ANALYSIS OF XIL QUERY LANGUAGE

As part of the XML query language survey in Chapter 2., an introduction to XIL was provided, briefly examining its features, syntax and data model. This chapter elaborates on these aspects (features, syntax and data model) further with the purpose of researching the system design and prototyping of a XIL based query system. This analysis is also used for possible extension to XIL research carried out by Junkkari et al. [2006].

3.1 XIL Syntax

As stated earlier in Chapter 2.7, XIL syntax is based on SQL where the select expression specifies the target of the query and from-where expression specifies the source of the query. There is only one ‘select’ keyword in a XIL query and is the only mandatory clause.

3.1.1 Formal Syntax

Junkkari et al. [2006] defines the syntax for XIL in terms of a context-free grammar [Knuth, 1968]. Using this grammar, the syntax is expressed as a finite set of productions of the form:

\[ X \rightarrow \alpha, \]

where, X is a single non-terminal symbol and \( \alpha \) is a string of terminal and/or non-terminals.

This grammar can be extended using meta-languages to describe the allowable structure and compositions of various syntactical expressions. This thesis uses EBNF [ISO/IEC 14977: 1996(E)] [Pattis R.] notation to represent the grammar of XIL syntax. A proposal for the XIL syntax using EBNF notations with explanation is as illustrated below.

<table>
<thead>
<tr>
<th>Sl.</th>
<th>XIL Syntax</th>
<th>Description of terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>QueryBody::=&quot;select&quot; SelectClause{&quot;&quot;,&quot; SelectClause} {FromWhereClause};</td>
<td>Each query begins with a select keyword. There can be one or more SelectClause or FromWhereClause attached with</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2.</td>
<td><strong>SelectClause::=</strong> [“each”] (SearchTerm) [AboutClause];</td>
<td>SelectClause can begin with an optional each keyword to reconstruct the view presented to the user. The mandatory value in this clause is the SearchTerm to specify the term being searched. The content-only query associated with the SearchTerm can be specified in the AboutClause.</td>
</tr>
<tr>
<td>3.</td>
<td><strong>AboutClause::=</strong> “about” CO@Query;</td>
<td>This clause is preceded with the about keyword followed by the Content-Only query (CO-Query) term.</td>
</tr>
<tr>
<td>4.</td>
<td><strong>FromWhereClause::=</strong> (FromClause) [(WhereClause)];</td>
<td>FromClause is the mandatory in FromWhereClause. It can also contain WhereClause.</td>
</tr>
<tr>
<td>5.</td>
<td><strong>FromClause::=</strong> “from” (SearchTerm) [(AboutClause)];</td>
<td>FromClause starts with the from keyword followed by the SearchTerm and having optional AboutClause.</td>
</tr>
<tr>
<td>6.</td>
<td><strong>WhereClause::=</strong> “where” (SearchTerm) [(AboutClause)];</td>
<td>WhereClause starts with the where keyword followed by the SearchTerm and having optional AboutClause.</td>
</tr>
<tr>
<td>7.</td>
<td><strong>SearchTerm::=</strong> [Separator] Step { Separator Step} [//];</td>
<td>A SearchTerm contains one or more Step, each of them could be preceded/followed by a Separator</td>
</tr>
<tr>
<td>8.</td>
<td><strong>Separator::=</strong> “/”</td>
<td>“//”;</td>
</tr>
<tr>
<td>9.</td>
<td><strong>Step::=</strong> “SearchWord”</td>
<td>“*”;</td>
</tr>
<tr>
<td>10.</td>
<td><strong>CO-Query::=</strong> k</td>
<td>phrase;</td>
</tr>
<tr>
<td>11.</td>
<td><strong>Phrase::=</strong> k ( “” k );</td>
<td>phrase contains two or more k, search keys, separated by space.</td>
</tr>
</tbody>
</table>
In Table 4, \textit{SearchWord} is any valid XML element name and \textit{CO-Query} is a valid content only (CO) query expression. (*\textit{SearchWord} and \textit{k} are search keys formed of allowed characters*).

### 3.1.2 Syntax Examples

This section formulates some sample queries based on the EBNF notation and syntax described in Table 4. XIL uses the power of path expressions to evaluate structure based queries. These expressions are similar to those in XPath explained in Chapter 2.3; certain query samples are as described below. Note that reserved words in XIL are capitalized for ease of reading.

a) **SELECT */**

This query uses only path expression to retrieve all the leaf elements from the input collection. As shown, the input collection is not explicitly mentioned in the formulated query as this is set in the background.

The next query focuses on retrieving titles of articles that are about the topic "object-oriented". Along with the structural condition of retrieving the titles of articles this query also has an additional condition related to content. The result should contain only titles that refer to the topic "object-oriented". XIL uses the \texttt{about} clause to specify the search condition for retrieving the content. These content query terms could be specified between "" or ' to be able to distinguish them from the reserved words of XIL.

b) **SELECT /collection//title ABOUT Object-Oriented**

When a content-only search condition is given, the elements in the XML collection that are relevant for the content-only condition should be marked and also the weights for these marked elements with respect to this search condition should be evaluated. XIL does not single out any particular mechanism for this but allows any weighting mechanism to be used. This study uses TRIX (Tampere retrieval and indexing system for XML) to evaluate
the weight of the terms related with the about clause. TRIX is a retrieval system that processes the CO-query and ranks the elements based on relevance [Kekäläinen et al., 2004].

c) SELECT author, /collection/title ABOUT Object-Oriented

To retrieve multiple elements from the query, in this case the author and article titles from the collection, the terms in the can be separated by a comma (,)

XIL uses the from clause to specify the source of the query. The XML collection considered can have p elements within abstract, author_biography and sec elements. To restrict retrieval of p elements from author_biography the query can be formulated as follows.

d) SELECT p FROM author_biography

In XIL queries can be formulated attaching where clause to from clause. The where clause is used to provide additional condition to the source element specified in the from clause. The following query retrieves p elements from articles that have sub-sections about programming logic.

e) SELECT p FROM article WHERE sub_sec ABOUT programming logic

The XML collection on which the XIL query is executed is mentioned in Appendix B.

3.2 Goals of XIL

Chapter 2.7.2 enumerated the XIL features. This section examines the various features in detail.

Declarative query formulation: The select-from-where-about syntax of XIL is based on SQL syntax. The expected result from XIL is specified in the select clause and the source of the query is specified in from-where clause. Content related searches are included in about clauses. This enables one to express queries of varying complexity easily
following the syntax specified in Table 4. It can also be observed from the XIL syntax that
the query language specifies what the output of the query should be and not how the output
should be obtained; making it declarative. Thus, the features of SQL such as its simplicity
and declarative query formulation are also passed over to XIL.

Support for queries based on content and structure or content only: The about clause,
mentioned in EBNF rule 3 [Table 4], is used for querying the content of the XML document.
Query can also be formulated using only about clause as given:

SELECT * ABOUT object-oriented

In this case, the retrieved results consist of all elements from the XML collection that include
the topic ‘object-oriented’. The result also includes the weights for these elements based on
the topic ‘object-oriented’. The about clause can also be combined with the select, from
or where clause and in these cases, it can query the content as well as the structure, as
given in query b) and c) [Chapter 3.1.2].

Support for querying truly heterogeneous XML collections: The processing of XIL query
uses a data model that is based on Dewey’s indices of the underlying XML documents.
This makes XIL processing independent of the underlying DTD or XML Schema [W3C XSD,
2008] definitions. The query processing of XIL can also contain elements such as * (any
element), / (child separator), // (descendent separator) in all valid combinations as shown in
EBNF rule 7 [Table 4Table 4]. These features ensure that XIL can query structured,
unstructured, valid or invalid collection of XML documents.

Views based on relevant XML elements: A view is a set of relevant elements of a document
which is presented to the user as the result of the query. Most data-centric query languages
display the content of the relevant elements as the result of the query. In this case, the
information about the structure of the document is lost. Document-centric query languages
like XQuery allow construction of views using FLWOR expressions. The advantage in using
Dewey-like indices (tree structure) is that it preserves the reading order of elements,
thereby, making it possible to build views. Junkkari et al. [2006] state various options for views in their study which are as summarized below:

1) Additional elements enclosing the result set: The retrieved query result is enclosed within `<result> </result>`. Each element in the result contains its corresponding Dewey indices and the element name. The reading order of the elements in maintained in this view but all information about ancestors is lost.

2) Maintain original document structure and show only relevant element: In this view original order of the document is maintained but the content is shown only for relevant elements. This view retains the information about the ancestors of the elements in the result set.

3) Maintain original document structure and show only most relevant element. This view is similar to the previous one with the exception that not all relevant elements are shown. The most relevant element displayed it that having the highest weight in the ancestor-descendant relation. Another option would be to display the element that has the highest common index among the relevant elements. This view displayed to the user can be extended using the document structure as Dewey’s indices retain this information.

### 3.3 Relevance Ranking in XIL

Chapter 2.7.4 introduced XIL data model and explained Dewey ordering in detail. This section explains the various aspects of the data model and the manner in which it helps to achieve relevance ranking in XIL. XIL data model or q-tree contains the name, index, relevance and weight of the elements in the XML collection. The XML collection should be modified to include additional attributes as given:

```
<name, index, weight, relevance>
```
The name of the element is the XML element name. The index or d_o property is the Dewey’s index of that element. Thus, the name of the element and its associated index are used to identify an element. The relevance attribute consist of Boolean values indicating whether that element is relevant for a given query or not. The initial value of relevance indicator is set to true for all elements indicating that all elements are relevant at the start of the query processing. As the various expressions are evaluated during the query processing all non-relevant elements are marked to false.

The weight attribute is computed when there is an about expression in the XIL query. For every about expression in the query, XIL interacts with the TRIX system. TRIX retrieves the XML documents based on the content of the about expression and ranks elements in the XML document based on their relevance. TRIX assigns weight to the element within the range \([0, 1]\). A non-zero weight of the element indicates the degree of relevance of an element and a zero weight indicates that the element is irrelevant.
Chapter 4. IMPLEMENTATION PROPOSAL FOR XIL

4.1 System Overview

4.1.1 High-level Architecture

Figure 2 illustrates the high-level architecture of the XIL query system. It consists of three main components as discussed below.

User Interface: Provides an interface for the user to input a XIL query. Once XIL core processes the query, the results are displayed on the result interface.

XIL Core Engine: This is the heart of the XIL processing. This engine carries out the daunting task of processing the valid input query for the XML collection. It interfaces south bound towards the TRIX system or any other relevance ranking data provider. Towards the north bound it receives the input query and returns the results. The core engine marks the relevant nodes according to the structural conditions specified in the query and processes the weights (of CO queries) returned from TRIX depending on the associated semantic.
**TRIX system**: TRIX system is external to XIL query system and is invoked to get the weights for the terms associated with the ‘about’ term. TRIX returns the weights associated with the node elements in the form of a simplified query tree (d-tree).

Figure 3 Sequence diagram for a XIL query with about clause

A high-level sequence diagram for a simple XIL query with an about clause is as shown in Figure 3. The data collection is shown as a separate entity in the above diagram merely to separate it out with the functions of XIL Core engine and TRIX.

The use-case sequence is as given below:

1) The user enters the XIL query into the user interface.

2) The XIL core engine processes the query towards the relevance ranking data provider (in this case the TRIX).
3) The XIL core engine receives the weighted tree structure which is restructured with relevance based weights and constructs a ranked result-view. The result is displayed to the user via the user interface.

4.1.2 Application Environment

The application environment used for prototyping is as given in Table 5 below:

Table 5 Application Environment

<table>
<thead>
<tr>
<th>Item Description</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>XIL Engine Operating system</td>
<td>Windows (Windows XP Professional) (or Linux)</td>
</tr>
</tbody>
</table>
| HW Specifications (minimum requirements) | CPU: 2 GHz  
RAM: 256 MB  
Disk Space: 1 GB |
| XIL UI + Core Engine | PHP5 based front end and PHP5 processing engine |

Various data types were considered while developing XIL which includes array of various dimensions, 2-D and 3-D, tree structure and DOM. Each of these data types was assessed for the support it could provide for processing XIL semantics. It was observed that DOM data model was most suitable as it represents the XML documents naturally as a tree structure and supports navigation in any direction. Hence, any language that provides the interface for DOM can be used for the implementation; e.g. PHP, Java.

PHP5 was used for implementation of the XIL prototype as it supports DOM API for operating on XML document. In a full-fledged implementation, this could be used together with Linux and apache to exploit the capabilities of LAMP (Linux, Apache, MySQL, PHP) stack in the future [Schott, 2005].

4.1.3 Interfaces

XIL query may contain both structure and content related conditions. To evaluate the content-only (CO) query term XIL core interfaces with an external system that provides the
weighting schema for the XML documents. In this study, TRIX has been chosen as this external interface.

TRIX system is an XML search engine that retrieves the XML documents based on the input query and ranks them based on their relevance [Kekäläinen et al., 2004]. This provides the weights for the XML document for each about clause (CO query term) in the XIL query. The data structure retrieved from TRIX is explained in detail in Chapter 4.2.2.2. The weights returned from TRIX are processed by the XIL core while evaluating the corresponding structure-based conditions.

Figure 4 Query Processing Flow across Interfaces

To explain this with an example, consider a XIL input query of the form: "select <e@clause> about <a@clause> from <e@clause> about <a@clause> where <e@clause> about <a@clause>". Here, the XIL core engine processes the select, from, where and about clauses while each content-only parameter following about is passed to TRIX to retrieve the weights for the elements. TRIX evaluates the content-only parameter, processes it and returns the relevance based ranking for a given xml collection. The core engine now uses the relevance weights and processes it based on structure (select, from and where). This is as illustrated in Figure 4.
4.2 Implementation Proposal

The section describes in detail the various factors that were taken into consideration while developing the algorithm for XIL.

4.2.1 User Interface

While designing the user interface two options were considered. The first being a single text field to input the query and the other option has multiple fields to input the query. In the following section both the options are discussed and their pros and cons are stated.

4.2.1.1 Single Text Field Option

The user interface in this case consists of a single input field that the user can input the XIL query and the result appears below text-area as shown in Figure 5.

![Figure 5 Simple UI for XIL Query](image)

This is similar to a simple Google search user interface design thereby providing a sense of familiarity to most users. Unlike queries input by users on the Google search, XIL query has a rigid select-from-where structure that the user needs to formulate. Though XIL has SQL-like statements most users would prefer to have cues to help formulate queries.
In this case, the query validation checks the structure of the query and the spellings of the reserved words (e.g. `select`, `from`, `where`). The query is read from the first reserved word (select statement) until the next reserved word and the terms in between are associated with the starting reserved word. This process is repeated till the end of the query.

### 4.2.1.2 Multiple Text Fields Option

In multiple input field approach, the user interface is split into separate `select` and `from-where` rows as shown in Figure 6.

![Figure 6 XIL Query UI with Multiple Query fields example](image)

Here, the user can also add extra `select` or `from-where` rows depending on the query they wish to formulate. The structure of the interface provides guidelines even for first time
users to be able to construct valid XIL queries with relative ease. This would ensure that queries formulated are structurally correct and validating them is relatively simpler as the user input is minimal. This in turn also makes the implementation of the XIL logic easier as the query is already broken down into different select & from-where clauses. Hence, finding the associated elements for each clause is easy because of the structure of the interface. The major advantage of this approach is that it does not require the user to know the exact syntax of the language. The ‘SELECT’ clause followed by a menu for the ‘EACH’ option and then a text box to enter the path expression. This is then followed by a menu which has all other feasible options such as ABOUT, FROM, WHERE. There are options to add more fields, for example, to add more select options or to add more from-where options as shown in Figure 6.

When developing the prototype for XIL the simple approach in 4.2.1.1 was chosen. For a complete implementation of XIL query language user interface, the multiple query fields approach enhances usability and reduces user errors.

### 4.2.2 XIL Core Logic

The syntactic analysis is simplified due to the multiple text input but the input by the user would have to be verified. This topic is not covered in this thesis and it assumes that the user has input a valid XIL query.

The query is then broken down into different clauses and the core logic consists of analyzing each clause and then combining them together to give the resultant view. The next task then lies in designing the underlying data structure.

#### 4.2.2.1 XIL Data Structures

Query Tree or q-tree is the most important data structure in XIL. Its main parts are Dewey’s indices (d_o or dewey_ord) for the XML content, the names of the element, the weights and relevance associated with each element. The default weights are set to zero and as all
nodes are considered as relevant the relevance value is set to Boolean value true. Table 6 provides the structure of a q-tree for XIL at the start of the query processing for the XML sample given in Appendix B. (Table 18).

Table 6 Structure of the q-tree for XIL

```xml
<collection>
  <article d_o="1" wt="0" relevance="true">
    <title d_o="1.1" wt="0" relevance="true"/>
    <author d_o="1.2" wt="0" relevance="true"/>
    <author d_o="1.3" wt="0" relevance="true"/>
    <author d_o="1.4" wt="0" relevance="true"/>
    <abstract d_o="1.5" wt="0" relevance="true"/>
    <index_term d_o="1.6" wt="0" relevance="true"/>
    <sec d_o="1.7" wt="0" relevance="true">
      <title d_o="1.7.1" wt="0" relevance="true"/>
      <p d_o="1.7.2" wt="0" relevance="true"/>
      ...
    </sec>
  </article>
  ...
  <article d_o="2" wt="0" relevance="true">
    <title d_o="2.1" wt="0" relevance="true"/>
    ...
  </article>
  ...
</collection>
```

4.2.2.2 TRIX Data Structures

Document Tree or d-tree is the principal data structure of TRIX server. It contains the indexed document structure to which inverted file can be attached. When the query contains ‘about’ clause the TRIX server is connected to gets the weights related to the content-only terms. TRIX prototype implementation can operate in two modes, online and batch. The
The current implementation of TRIX would have to be modified to produce the d-tree as required for further processing by XIL core. The d-tree would consist of indices for the elements in the XML collection and the weights for all elements with respect to the `about` clause.

TRIX online mode performs the search on the default database that is linked. This database should be modified to point to the XML collection that is used in the XIL query. Batch mode can also be used if the `about` terms in the XIL query are known in advance. This restricts the user while formulating the XIL query and hence was not considered appropriate to be used while processing XIL queries.

### Table 7 Example for constructing the d-tree returned from TRIX

<table>
<thead>
<tr>
<th>Dewey's ordering</th>
<th>Elem</th>
<th>about</th>
<th>Deductive Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>article</td>
<td>0.3</td>
<td>X</td>
</tr>
<tr>
<td>1.1</td>
<td>title</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>abstract</td>
<td>0.1</td>
<td>X</td>
</tr>
<tr>
<td>1.3</td>
<td>index_term</td>
<td>0.1</td>
<td>X</td>
</tr>
<tr>
<td>1.4</td>
<td>sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4.1</td>
<td>title</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4.2</td>
<td>p</td>
<td>0.1</td>
<td>X</td>
</tr>
<tr>
<td>1.5</td>
<td>sec</td>
<td>0.2</td>
<td>X</td>
</tr>
<tr>
<td>1.5.1</td>
<td>sub_sec</td>
<td>0.2</td>
<td>X</td>
</tr>
<tr>
<td>1.5.1.1</td>
<td>p</td>
<td>0.1</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>article</td>
<td>0.1</td>
<td>X</td>
</tr>
<tr>
<td>2.1</td>
<td>title</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>abstract</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>index_term</td>
<td>0.1</td>
<td>X</td>
</tr>
<tr>
<td>2.4</td>
<td>author_biohistory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4.1</td>
<td>p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4.2</td>
<td>p</td>
<td>0.1</td>
<td>X</td>
</tr>
<tr>
<td>2.5</td>
<td>author_biohistory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.5.1</td>
<td>p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.5.2</td>
<td>p</td>
<td>0.1</td>
<td>X</td>
</tr>
</tbody>
</table>
Table 7 is an example in which the weights are marked for the sample d-tree for the about term ‘Deductive Database’. This is extended from the work carried out by Junkkari et al [2006]. The Dewey ordering, weights and relevance mentioned here is hypothetical.

The d-tree returned from TRIX, consists of Dewey indices for only those elements that have some weight associated with them. The element name and relevance value field are omitted in this case to avoid duplication of information.

Table 8 Structure of d-tree for sample XML

```
<d_o="1" wt="0.3">
  <d_o="1.2" wt="0.1"> </>
  <d_o="1.3" wt="0.1"> </>
  <d_o="1.4">
    <d_o="1.4.2" wt="0.1"> </>
  </>
  <d_o="1.5" wt="0.2">
    <d_o="1.5.1" wt="0.2">
      <d_o="1.5.1.1" wt="0.1"> </>
    </>
  </>
</>

<d_o="2" wt="0.1">
  <d_o="2.3" wt="0.1">
    <d_o="2.4">
      <d_o="2.4.2" wt="0.1"> </>
    </>
  </>
  <d_o="2.5">
    <d_o="2.5.2" wt="0.1"> </>
  </>
</>
```

The structure of d-tree, shown in Table 8, is that of a simplified q-tree so that common operations like addition of weights can be performed over both the data structures.

4.2.3 Semantic rules

The semantic rules for processing the different clauses in XIL are expressed using attribute grammar [Junkkari et al., 2006]. These rules are defined by inherited and synthetic attributes, processed as a set of indices. The indices of the underlying collection are defined using Dewey’s indices. A semantic rule specifies the operation that has to be performed on
the set of indices for the associated query clause. When processing a XIL query the evaluation starts with applying the rule for the last clause in the query. The result obtained for this last clause is passed as input when processing the previous clause. These steps are repeated until the rule associated with select clause is carried out. The resulting set of indices thus obtained has the weights and relevance computed based on the input query.

The semantic rules thus provide a mechanism of obtaining the valid indices and weights for a clause. For any path expression or e-clause in Figure 4, the rule is to set all nodes similar to this element as true and set all other nodes as false. When evaluating a where clause all nodes that are already marked as relevant are set as irrelevant and the parent of these elements is set as relevant; weight component is not altered during the evaluation of this clause. When evaluating from clause for all nodes that are relevant, its children nodes are set as relevant and the weights of the parent node is transferred to each child node; the parent node is then set as irrelevant and their weight is set to be zero. During this evaluation the weights are set to zero for any node that was marked as irrelevant from earlier processing.

4.2.4 XIL Algorithm

XIL algorithm is explained in detail in this section. The section first explains the background work carried out to arrive at the chosen XIL algorithm. The next section details XIL algorithm, assumptions made and provide the pseudo-code.

4.2.4.1 Background

A study formulating the actual queries to determine the order in which the clauses should be processed and effectively use the predefined data-structure was carried out. The dry run details performed as part of this study is stored in Appendix A. .

The first method is based closely to the steps detailed by Junkkari et al. [2006]; when evaluating an about clause the calculation of weights uses a simple addition operation.
This method uses a single q-tree data structure when evaluating different clauses. The next method carried out for all the input queries follows similar rules as the earlier method but a separate q-tree data-structure is assigned for processing every line on the user interface. A XIL query with multiple from-where clause would assign a separate q-tree for every from-where clause and a combined operator used to give a single q-tree and this would be combined with the q-tree for select clause to give the result. The benefit of this method is that different combining operators can be implemented for different clauses making it possible to give additional weight to the target of the query than the source. The dry run elucidates this by considering two options when evaluating the about clause; the weights are assigned depending on the relevance of the element.

The remaining methods in the dry run validate some query optimizing criteria that can be used during implementation. These include setting only the irrelevant nodes when evaluating the e-clause for relevance or when evaluating from clause, the children nodes are not processed if the parent node is irrelevant. The dry run also cites weighting mechanism that should be avoided as they mark irrelevant elements as relevant or vice-versa.

### 4.2.4.2 Chosen Solution

The dry run was carried out based on a few assumptions that hold for all the method that were carried out. The initial q-tree data-structure is defined to have all nodes marked as relevant and weight set to zero. The order of processing an each clause attached to the e-clause element, in a select clause, is altered from the original semantic rules for simplicity. The each clause is the last clause to be evaluated in the XIL query, after the select clause, and this could alter the weights shown in the result-view.

From the result of the dry run it was noticed that in XIL queries having multiple from-where clauses the order of processing these clauses affected the result. To have a standard behavior it has been assumed that a from clause connected to an e-clause with the least
scope would be evaluated first followed by the from clause with the higher scope clause. An example of this is a XIL query with three from-where clause related to sec, subsec and p element for a collection adhering to the DTD given in Table 1. When evaluating the query the algorithm should first evaluate the from clause related to p first then followed by the subsec and finally for the sec element.

The first method explained in the previous section was used when deriving the XIL algorithm used for implementation. This pseudo-code for the different clauses is outlined in Appendix C. for reference. In this method when evaluating the select clause for a XIL query having an about clause the processing sets nodes that have no weights but being relevant and also sets nodes that are not relevant but have some weights associated with it as irrelevant. Thus for a query having an about clause the weights and the relevance attributes are considered when the result set is created. In XIL queries without the about clause the weight attribute is ignored when the select clause is processed.
Chapter 5. COMPARING XIL TO EXISTING QUERY LANGUAGES

This chapter compares and contrasts on the researched query languages with XIL focusing on the query formulation and result. To perform this, sample queries from existing query languages are formulated and results are shown. XIL queries are also formulated for the same task, where applicable. The XML collection that is being queried is based on the DTD in Table 1 (articles.xml) and is given in Appendix B.

5.1 XPath: Examples and Result

The following XPath queries have been formulated and the corresponding result for the XML collection considered is given below. For comparison XIL queries are formulated corresponding to the XPath queries.

<table>
<thead>
<tr>
<th>Sl.</th>
<th>Purpose of query</th>
<th>Query and Result</th>
</tr>
</thead>
</table>
| 1   | To retrieve the titles for all articles in the collection | /collection/article/title  
XIL: SELECT /collection/article/title  
SELECT article/title FROM collection³  
&lt;title&gt;Logical Foundations of Object-Oriented and Frame-Based Languages&lt;/title&gt;  
&lt;title&gt;Part-Whole Relationship Categories and Their Application in Object-Oriented Analysis &lt;/title&gt;  
&lt;title&gt;Extending SQL with Generalized Transitive Closure&lt;/title&gt; |
| 2   | Retrieve all the titles nodes in the collection | /collection/title  
XIL: SELECT /collection/title  
SELECT title FROM collection⁴  
&lt;title&gt;Logical Foundations of Object-Oriented and Frame-Based Languages&lt;/title&gt;  
&lt;title&gt;Introduction&lt;/title&gt;  
&lt;title&gt;A Perspective on Object-Oriented LW. Declaration Programming&lt;/title&gt;  
&lt;title&gt;F-Logic by Example&lt;/title&gt; |

³ This is an alternative XIL query for SELECT /collection/article/title
⁴ This is an alternative XIL query for SELECT /collection/title
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>3</strong></td>
<td><strong>Node test to find the text element from the XML document</strong></td>
</tr>
<tr>
<td></td>
<td><code>/collection/article/author_biography/text()</code></td>
</tr>
<tr>
<td></td>
<td>XIL: <code>SELECT /collection/article/author_biography/text()</code></td>
</tr>
<tr>
<td></td>
<td>Renate Motschnig-Pitrik received her doctorate in ......</td>
</tr>
<tr>
<td></td>
<td>Jens Kaasbøll received his PhD degree in ......</td>
</tr>
<tr>
<td></td>
<td>Shaul Dar received the BSc. degree in ......</td>
</tr>
<tr>
<td></td>
<td>Rakesh Agrawal received the M.S. and ......</td>
</tr>
<tr>
<td><strong>4</strong></td>
<td><strong>Query to conditional select the nodes</strong></td>
</tr>
<tr>
<td></td>
<td><code>/collection/article[title='Extending SQL with Generalized Transitive Closure']/author</code></td>
</tr>
<tr>
<td></td>
<td>XIL: <code>SELECT author FROM /collection/article WHERE title ABOUT 'Extending SQL with Generalized Transitive Closure'</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;author&gt;Shaul Dar&lt;/author&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;author&gt;Rakesh Agrawal&lt;/author&gt;</code></td>
</tr>
<tr>
<td><strong>5</strong></td>
<td><strong>XPath query with contains clause to display all authors from titles containing the term ‘extending SQL’</strong></td>
</tr>
<tr>
<td></td>
<td><code>/collection/article/title[contains(.,’extending SQL’)]/author</code></td>
</tr>
<tr>
<td></td>
<td>XIL: <code>SELECT author FROM /collection/article WHERE title ABOUT 'extending SQL'</code></td>
</tr>
<tr>
<td></td>
<td>No results displayed for XPath query as it does string comparison.</td>
</tr>
<tr>
<td></td>
<td>Result for XIL query is marked is shown in bold:</td>
</tr>
<tr>
<td></td>
<td><code>&lt;author&gt;Shaul Dar&lt;/author&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;author&gt;Rakesh Agrawal&lt;/author&gt;</code></td>
</tr>
</tbody>
</table>
The search results using XPath indicate that it is efficient at retrieving the node data; making it suitable to retrieve information based on structural related conditions. It provides support for content related queries with the `contains()` function for string comparison. It can only be used on one XML document and no relevance ranking is supported. The result in XPath is a list of valid nodes without maintaining the XML document structure.

The queries in XPath can be embedded into other XML query languages to support retrieval from a document-centric collection. This allows the use of XPath expressions in structure related clauses in a XIL query. From an implementation perspective, programming languages like Java, C++, PHP etcetera support XPath which enables the use of XPath interfaces easily in XIL.

From Table 9, it can be noted that XPath queries can be easily expressed as XIL queries. The main difference lies in the presentation of the result-view. For the XPath queries 1 to 4 in Table 9 the given XIL queries would typically display the entire structure of the XML document but showing content for only the relevant elements. The structure of the XML document is retained, thereby enhancing the readability of the result-view. Query 5 in Table 9 is an example of XPath `contains()` function which returns true if the first argument string contains the second argument string or otherwise returns false [W3C XPath 1.0, 1999]. The `contains` function performs a string match and hence the result for the XPath query is empty because of case mismatch. In contrast, the XIL query displays the author names for the title matching the `about` clause. XIL queries 4 and 5 also demonstrate that the queries can be given without nested query structures.

5.2 XIRQL: Examples and Result

XIRQL queries are formulated in Table 10, highlighting its key features like weighting, structural vagueness and data-types with vague predicates. The index nodes selected are
The definitions of p, sec and article and these definitions are part of the XML Schema. The XIL queries corresponding to the given XIRQL are formulated where applicable.

### Table 10 XIRQL queries and results

<table>
<thead>
<tr>
<th>Sl.</th>
<th>Purpose of query</th>
<th>Query and Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Query to display all p elements that contain the words 'programming' and 'logic'.</td>
<td>XIL: SELECT p FROM /collection/article//section WHERE p ABOUT 'programming logic'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A number of researchers argued that object-oriented languages are fundamentally different from (and even incompatible with) other paradigms, especially with logic programming [Maier 1989; Unman 1987]. This point of view was reflecting disappointment with the lack of early success in formalizing a number of central aspects.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>......</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The detailed result for this query is given in Appendix D.</td>
</tr>
<tr>
<td>2</td>
<td>The query returns all elements and attributes with the name author.</td>
<td>XIL: Not supported but possible to extend</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MICHAEL KIFER\nGEORG LAUSEN\nJAMES WU\nRenate Motschnig-Pitrik\nJens Kaasbøll\nShaul Dar\nRakesh Agrawal</td>
</tr>
<tr>
<td>3</td>
<td>Query to retrieve element names that are similar to each other, for example abstract can be abs or abstract_text in the XML.</td>
<td>XIL: Not feasible</td>
</tr>
<tr>
<td></td>
<td></td>
<td>We propose a novel formalism, called Frame Logic ......</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DPart decomposition and, conversely, the ......</td>
</tr>
<tr>
<td></td>
<td></td>
<td>We present SQLiTc, an extension ......</td>
</tr>
<tr>
<td>4</td>
<td>This query retrieves all titles in the XML; but ranks the titles of article elements higher than titles of section or subsection elements. Thus the elements marked</td>
<td>XIL: Not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Logical Foundations of Object-Oriented and Frame-Based Languages\nIntroduction\nA Perspective on Object-Oriented LW. Declaration Programming</td>
</tr>
</tbody>
</table>
XIRQL uses the \texttt{cw} clause for testing the word occurrence in an element. Query 1 in Table 10 shows that this is similar to the \texttt{about} clause in XIL and hence the corresponding XIL query can be formulated. Queries 2 to 4 in Table 10 are examples of XIRQL providing support for vagueness that is a typical property of queries in IR. These features are not supported in XIL and the corresponding XIL queries are not given. However, some of these features can be extended to XIL and a note is made of this for future investigation.

Query 2 in Table 10, removes the distinction between attribute and element names in the XIRQL query by using the \texttt{=\_} before the name. This feature can be extended to XIL by handling it during query processing using the existing data-structure. Query 3 gives an example of the similarity operator that retrieves elements that are semantically similar to the element name preceded by \texttt{\~}. This feature is supported in XIRQL by extending the database schema definitions; hence this cannot be supported in XIL by the existing data model. Query 4 is an example of XIRQL generalizing the parent-child relationship by using the \texttt{\_\_} operator. This is a useful feature when the precise path to an element is not known but the user would like to identify the most relevant element in the result; these are marked in bold in the result.

To support the various features mentioned in XIRQL, XML Schema is used to describe the document structure and the data-type of the elements. Thus, XIRQL queries are restrictive.
as it only process XML documents confirming to an XML Schema. In contrast, XIL is unaware of the underlying DTD or XML Schema definitions and it cannot support features that use these data-definitions for IR. When investigating the features of XIRQL, it was noted that some features are not dependant on the XML Schema. For example, generalizing parent-child relationship using \ or removing distinction between attributes and elements using = could be extended in XIL.

5.3 XQuery: Examples and Result

In this section, we cite a few concrete samples for XQuery and provide the corresponding XIL queries, where feasible. The power of XQuery lies in the FLWOR expression and a couple of examples of this are considered below.

Table 11 XQuery queries and results

<table>
<thead>
<tr>
<th>Sl.</th>
<th>Purpose of query</th>
<th>Query and Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>To extract the title of articles where the author is Shaul Dar</td>
<td>( \text{doc(&quot;articles.xml&quot;)/collection/article[author='Shaul Dar']/title} )</td>
</tr>
</tbody>
</table>

Extending SQL with Generalized Transitive Closure

| 2   | FLWOR expression is used to extract the introduction paragraph from the articles. The resultant nodes are ordered based on the title of the articles followed by the content of the introduction section for each article. | for $x \text{ in doc("articles.xml")//article}$y := $x/sec/title where $y \text{ eq ’INTRODUCTION’}$ order by $x/title return <introduction>{ $x/title , $y/p }</introduction> | XIL : SELECT title, p FROM //article WHERE sec/title ABOUT ’Introduction’ |

<introduction>
  <title>Extending SQL with Generalized Transitive Closure</title>
  <p>THE ABILITY to process..............</p>
  <p>We are interested in.........</p>
  .......
</introduction>
47

<table>
<thead>
<tr>
<th>3</th>
<th>Use the powerful FLWOR expression to extract the titles of articles that have more than two authors.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>for $x$ in doc(&quot;articles.xml&quot;)//article</td>
</tr>
<tr>
<td></td>
<td>let $y := $x/author</td>
</tr>
<tr>
<td></td>
<td>where count($y) &gt; 2</td>
</tr>
<tr>
<td></td>
<td>return $x/title</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>XIL:</strong> No support for “count”</td>
</tr>
</tbody>
</table>

3. Use the powerful FLWOR expression to extract the titles of articles that have more than two authors.

4. In this query the conditional expression is used to give a result where the first two author names are given for each article and the remaining authors are combined under “et al.” [Robie J, 2003].

<table>
<thead>
<tr>
<th>4</th>
<th>In this query the conditional expression is used to give a result where the first two author names are given for each article and the remaining authors are combined under “et al.” [Robie J, 2003].</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>for $x$ in doc(&quot;articles.xml&quot;)//article</td>
</tr>
<tr>
<td></td>
<td>return</td>
</tr>
<tr>
<td></td>
<td>&lt;article&gt;</td>
</tr>
<tr>
<td></td>
<td>{ $x/title }</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>for $y$ at $i$ in $x/author</td>
</tr>
<tr>
<td></td>
<td>where $i$ &lt;= 2</td>
</tr>
<tr>
<td></td>
<td>return &lt;author&gt; $a &lt;author&gt;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>if (count($x/author) &gt; 2)</td>
</tr>
<tr>
<td></td>
<td>then &lt;author&gt; et al. &lt;author&gt;</td>
</tr>
<tr>
<td></td>
<td>else ()</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>&lt;/article&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>XIL:</strong> Not feasible</td>
</tr>
</tbody>
</table>

| <article>                                                                                                                                                                                                 |
| <title> Logical Foundations of Object-Oriented and Frame-Based Languages |
XQuery is semantically similar to SQL and is easy to use [Graaumans, 2005]. However, when comparing XQuery with XIL for this research, XQuery query formulation was found to be relatively difficult. XQuery allows users to reconstruct the result-view for a query using the return clause. The user is able to manipulate the data hierarchy in the result-view by incorporating conditional expression, arithmetic operator or string operators in the return clause. The user can, thus, construct complex views based on their programming skills.

Query 3 in Table 11 cannot be expressed as a XIL query as there is no support for the count function. Query 4 in Table 11 gives an example of XQuery where the result-view is constructed to show author as “et al.” when there are more than two authors for an article in the result. The corresponding XIL query for query 4 cannot be formulated as the user cannot manipulate the structure of the result-view as XIL constructs the result-view automatically for the user and free them from this task. The reading order of the XML collection is maintained and only relevant elements are displayed in the XIL result-view. This differs from the construction of views in XQuery where the user is responsible to provide the FLWOR return clause to get the desired result-view. However, if multiple
elements are returned by the XQuery, maintaining the reading order of the elements in the result-view could be challenging. It is also problematic to represent only valid elements in the result-view while maintaining the hierarchical structure of the XML collection.

For queries 1 and 2 in Table 11, the corresponding XIL queries are also given. The main difference is that XIL is not case sensitive as XQuery; hence the result-view for query 2 contains all three articles in the XML collection rather than just the two returned by XQuery. XQuery result-view can also be ordered using the order by clause which is not supported in XIL as it maintains the reading order of the XML collection.

5.4 TeXQuery: Examples and Result

The following TeXQuery queries give example of full-text searches embedded in XQuery expression and an example is also given where the TeXQuery result-list is reconstructed by embedding it in a FLWOR. The corresponding XIL queries are also given for these queries, where applicable. As TeXQuery data model allows XQuery expressions to be embedded in full-text search, an example of this is considered. The result of this query is a Boolean true or false hence it is not possible to formulate XIL query to get the same result.

Table 12 TeXQuery queries and results

<table>
<thead>
<tr>
<th>Sl.</th>
<th>Purpose of query</th>
<th>Query and Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Retrieve the title of article that have paragraph about logic and programming.</td>
<td>//article[.//p ftcontains 'logic' &amp;&amp; 'programming' with stems]/title</td>
</tr>
<tr>
<td></td>
<td>Stemming is used so program(s), programmer(s) are valid values for programming.</td>
<td>XIL: SELECT title FROM //article WHERE p ABOUT 'logic programming'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Logical Foundations of Object-Oriented and Frame-Based Languages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Extending SQL with Generalized Transitive Closure</td>
</tr>
<tr>
<td>2</td>
<td>Query to get all the paragraphs that have the terms 'logic' and 'programming'</td>
<td>//article[.//p ftcontains 'logic' &amp;&amp; 'programming' with stems window 2 without stopwords]/p</td>
</tr>
<tr>
<td></td>
<td>with stems and at a distance of 2 between them.</td>
<td>XIL: 'window 2' not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A number of researchers argued that object-oriented languages are fundamentally different from (and even incompatible with) other paradigms, especially with logic programming [Maier 1989; Unman 1987]. This point of view</td>
</tr>
</tbody>
</table>
was reflecting disappointment with the lack of early success in formalizing a number of central aspects.

......

The detailed result for this query is given in Appendix D.

<table>
<thead>
<tr>
<th>3</th>
<th>This query retrieves the similar paragraph content returned in previous query. The result-view is reconstructed to shown article, followed by its title and then all the paragraph elements. This is achieved by embedding the TeXQuery in an XQuery.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For $x$ in doc(&quot;articles.xml&quot;)//article let $y := $x[.//p ftcontains 'logic' &amp; 'programming' with stems window 2 without stopwords] return &lt;article&gt;{ $x/title , $y/p }&lt;/article&gt; XIL: Same as 1 but no support for ‘window 2’</td>
</tr>
<tr>
<td></td>
<td>&lt;article&gt; &lt;title&gt; Logical Foundations of Object-Oriented and Frame-Based Languages&lt;/title&gt; &lt;p&gt; A number of researchers argued that object-oriented languages are fundamentally different from (and even incompatible with) other paradigms, especially with logic programming ...... &lt;/p&gt; &lt;/article&gt;</td>
</tr>
<tr>
<td>4</td>
<td>This query returns true as there is an article in the collection with the author as ‘Shaul Dar’ This TeXQuery has an XQuery embedded in it.</td>
</tr>
<tr>
<td></td>
<td>//collection ftcontains //article[./author='Shaul Dar'] any XIL: Not feasible TRUE</td>
</tr>
</tbody>
</table>
TeXQuery can express complex full-text search by using Boolean predicates (like and, or not), stemming, limiting the scope of the search (within a node, sentence or paragraph) and specifying the window size. The TeXQuery queries given in Table 12 use the dot (.) operator to indicate the current context, for example //article[.//p] searches for an article containing p element as descendant. Query 1 and 2 in Table 12 use some of these conditions when formulating the TeXQuery. Query 1 can also be expressed as XIL query as support for these features are available. However, specifying the window size is not supported currently by XIL. Hence, query 2 cannot be expressed as a XIL query. However, this feature does not require any changes to the XIL functionality and has to be handled in the TRIX server and can be taken up for future development. TeXQuery allows reconstructing the result-view by embedding it in FLWOR expressions. The result-view appears as specified by the return clause as given in query 3 in Table 12. This dependency on XQuery expression implies that the pros and cons of XQuery result-view reconstruction also hold for TeXQuery.

5.5 Summary of Comparison

This section summarizes the results of the comparisons carried out as part of this study. The first part of this comparison is with respect to the XIL goals specified in Chapter 3.2. These results are as shown in Table 13 to Table 16. Table 17 compares all fives languages considered in this study for general properties of query languages.

The five query languages considered in this study are declarative queries of varying degree as the syntax of each query language dicatates what the result is rather than specify how to obtain the result. XQuery has a procedural-feel as its syntax permits iterators, variables, functions and conditional expression. In contrast, XIL has high level of declarativity as it does not use variables, external functions or programming strutures when forumulating queries. Table 13 compares the query languages for their query formulation and degree of declarativity.
Table 13 Comparing query languages for query formulation

<table>
<thead>
<tr>
<th></th>
<th>Declarative query formulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPath</td>
<td>Medium, as external functions are allowed but not programming structures such as IF-THEN-ELSE.</td>
</tr>
<tr>
<td>XIRQL</td>
<td>Medium, as external functions are allowed but not programming structures.</td>
</tr>
<tr>
<td>XQuery</td>
<td>Low, as the syntax allows iterators, variables, functions and conditional expression such as IF-THEN-ELSE.</td>
</tr>
<tr>
<td>TeXQuery</td>
<td>TeXQuery is an extension of XQuery thus has the same level level of declarativity. Low, as the syntax allows iterators, variables, functions and conditional expression such as IF-THEN-ELSE.</td>
</tr>
<tr>
<td>XIL</td>
<td>High level of declarativity as the syntax is based on SQL and does not use variables or programming structures.</td>
</tr>
</tbody>
</table>

A content only query language allows only content-oriented condition without any restriction on the structure of XML collection. However, content and structure query language allows the query to specify the structure and content related conditions. XPath and XQuery provide support only for string matching. Hence, a case mismatch would not bring the same result as XIL. In contrast, document-centric query languages such as XIRQL, TeXQuery and XIL search for content only terms with stemming. Table 14 compares the query languages for the support it provides for content and structure or content only queries.

Table 14 Comparing query languages for handling CAS and CO queries

<table>
<thead>
<tr>
<th></th>
<th>Support for CAS and CO queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPath</td>
<td>Yes, using contains() clause that supports only string comparison</td>
</tr>
<tr>
<td>XIRQL</td>
<td>CO supported with the cw clause</td>
</tr>
<tr>
<td>XQuery</td>
<td>String comparison provided with the value comparison operator eq</td>
</tr>
<tr>
<td>TeXQuery</td>
<td>CO supported with the ftcontains clause</td>
</tr>
<tr>
<td>XIL</td>
<td>CO supported with about clause</td>
</tr>
</tbody>
</table>
Table 15 compares the query languages for its ability to query truly heterogeneous collection without the user being aware of the underlying XML collection structure.

**Table 15 Comparing query languages for handling heterogeneous collection**

<table>
<thead>
<tr>
<th>Support for truly heterogeneous collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xpath</td>
</tr>
<tr>
<td>Yes. Heterogeneous collection can be queried using wild card operators such as <em>,/</em>,// *</td>
</tr>
<tr>
<td>XIRQL</td>
</tr>
<tr>
<td>No. The XML collection must conform to an XML Schema for processing.</td>
</tr>
<tr>
<td>XQuery</td>
</tr>
<tr>
<td>Yes.</td>
</tr>
<tr>
<td>TeXQuery</td>
</tr>
<tr>
<td>Yes.</td>
</tr>
<tr>
<td>XIL</td>
</tr>
<tr>
<td>Yes, supports querying XML collection being unaware of the underlying structure.</td>
</tr>
</tbody>
</table>

XIRQL uses XML Schema to define the index nodes and data types of elements to permit structural vagueness while querying, as discussed in Chapter 5.2.

Table 16 compares query languages for the relevance based views presented to the user as the result. Construction of views is a salient feature of XIL as it automatically constructs the views for the elements mentioned in the `select` clause while retaining the structure of the XML collection.

**Table 16 Comparing query languages for relevance based views**

<table>
<thead>
<tr>
<th>Views based on relevant elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xpath</td>
</tr>
<tr>
<td>Displays retrieved element from the original document</td>
</tr>
<tr>
<td>XIRQL</td>
</tr>
<tr>
<td>Uses a subset of the XQuery FLWOR to display the result.</td>
</tr>
<tr>
<td>XQuery</td>
</tr>
<tr>
<td>Allows restructuring of the view with FLWOR <code>return</code> clause. However, the queries are not simple to formulate and maintaining the reading order of the XML collection would be challenging task.</td>
</tr>
<tr>
<td>TeXQuery</td>
</tr>
<tr>
<td>TeXQuery embedded in XQuery FLWOR statements display the relevant elements by restructuring the view. Thus, the shortcomings of XQuery when constructing the result-view are handed over to TeXQuery.</td>
</tr>
<tr>
<td>XIL</td>
</tr>
<tr>
<td>Constructs the result-view by displaying the only the relevant elements while maintaining the reading order of the XML collection.</td>
</tr>
</tbody>
</table>
Table 17 summarizes the main finding from the comparison of XML query languages.

Table 17 Summary of comparison of XML query languages

<table>
<thead>
<tr>
<th>Feature</th>
<th>XPath</th>
<th>XIRQL</th>
<th>XQuery</th>
<th>TeXQuery</th>
<th>XIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weighting</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Boolean Retrieval</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Relevance Oriented search (irrespective of structure)</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Similarity Operator</td>
<td>String matching</td>
<td>Content matching with stemming</td>
<td>String match</td>
<td>Content matching with stemming</td>
<td>Content matching with stemming</td>
</tr>
<tr>
<td>Extensible type hierarchy for different data types (stemming, sub-string matching)</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Structural Relativism (drop distinction between attribute/element, exploit ontology over element names )</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Can be extended for features independent of XML Schema or DTD.</td>
</tr>
<tr>
<td>View Formulation</td>
<td>Show retrieved elements in document</td>
<td>As a subset of XQuery FLWOR clause</td>
<td>Restructuring of result using FLWOR</td>
<td>Embedded in XQuery</td>
<td>Automatic construction of view</td>
</tr>
</tbody>
</table>
Chapter 6. CONCLUSIONS

6.1 Summary

The growing amount of XML data in the computing industry has continuously ensured development of faster and efficient retrieving tools. This thesis investigated the features of XIL and validated it using a prototype. The benefits of XIL include simple query formulation and choice to query any XML collection without being aware of its structure or underlying DTD or XML Schema. It was verified that XIL is able to query a truly heterogeneous XML collection by using Dewey’s indices when constructing the result. The result of a XIL query retains information of the Dewey’s indices, relevance and weights calculated for each element in the collection. This information allows manipulation of the result-view based on parent-child or ancestor-descendant relationship for example grouping all elements from the same article based on relevance. From an IR perspective, this enables effective XML information retrieval with relevance ranking. A major drawback of using only Dewey’s indices in XIL is that it cannot utilize the data-type information stored in DTD or XML Schema to support various structural vagueness operations as in XIRQL.

The study of four existing XML query languages, of different genre, provided a framework for comparing the features offered by XIL. The aim of this study was to highlight the strengths and shortcomings of these different languages that were developed with the purpose of answering some issues related to XML retrieval. In comparison with these query languages, XIL queries were relatively simple to formulate for users with some SQL knowledge. When developing the full implementation for XIL, it has been recommended to use the UI having multiple query fields. This UI facilitates users to formulate XIL queries with no prior SQL knowledge. Studying existing query languages revealed various tried and tested methods that could be borrowed when developing XIL. To give an example, the XIL prototype uses XPath PHP API when querying for e-clause elements. Also, XIL functionality can be easily extended to support enhanced IR features such as structural
vagueness by generalizing parent-child relationship, as in XIRQL. The major difference was found in the manner in which the result-view was reconstructed by XIL. The existing languages that allowed reconstructing of the result used XQuery FLWOR expressions were not simple to formulate. In contrast, XIL reconstructs the views in the background using Dewey’s indices, making the task simple for the user.

The results from this prototyping highlight that the features offered by XIL are worthy of developing it further into a full-fledged application library.

6.2 Lessons Learned

The lesson learnt from developing XIL query language is that there are still numerous challenges faced when retrieving results from an XML, especially from an information retrieval perspective. There is still a need to develop an XML query language with the same stature as SQL has in relational database management systems. However, similar work carried out by numerous groups is a step in the right direction to achieving this means.

6.3 Recommendations for Further Work

The XIL prototype developed as part of this thesis caters for simple queries and to verify the processing of the main clauses in the query. The assumptions made when developing this prototype ensured that XIL queries following a particular pattern returned the correct relevant elements in the result-view. To extend this prototype to support XIL queries having multiple from-where or select clause would mean revisiting the core functionality of the XIL Engine. When developing the full implementation for XIL, the processing of the XIL Engine would most likely have to be carried out in a recursive manner; traversing from the head of the input query till the final e-clause element is met and then walking backwards marking relevant nodes based on query condition.

The input query validations carried out in the prototype are basic and restrictive from a user perspective. The XML elements should exactly match a hard-coded list to be processed by
the XIL Engine. Regular expressions can be used to validate the query input. The prototype can easily be extended to interface with TRIX or similar systems to calculate the weights for the about clause; the weight attribute is pre-populated for the relevant elements in the q-tree. The full implementation of XIL should interface with any system that can provide the weights for content-only search.

While the prototyping focused on simple reconstruction of result-views displaying attributes for relevant element alone, it is feasible to extend this for use in complex algorithms, in tandem with Dewey’s indices.

The prototype implementation of XIL is developed in PHP5. The core processing for query languages, like XPath interfaces, are developed in C. A similar approach could be taken for XIL once it is approved as a document-centric query language. Such endeavors could enable the extended use of XIL by the developer community, thereby making its use wider and development faster.
REFERENCES


APPENDICES

Appendix A. QUERY FORMULATION STUDY

An investigation was carried out to find the best method when using the different data structures. For this sample queries were considered and each method was walked through step by step. The details of this can be found in the attached excel sheet.

C:\\Users\MSc\\Thesis\XIL_Logic.xls
Appendix B. SAMPLE XML FOR QUERY ANALYSIS

The sample XML collection made for query analysis and processing is as given in Table 18. This collection while analyzing the various languages researched in this thesis. This is also used as input by XIL processing as well as in TRIX as the sample XML collection.

Table 18 Sample XML file, article.xml

```xml
<collection>
<article id="1">
<title> Logical Foundations of Object-Oriented and Frame-Based Languages </title>
<author> MICHAEL KIFER </author>
<author> GEORG LAUSEN </author>
<author> JAMES WU </author>
<abstract> We propose a novel formalism, called Frame Logic …… </abstract>
<index_term> H.2.1 [Database Management]: Languages…… </index_term>
<sec numbering="1">
<title> Introduction </title>
<p> In the past decade, considerable interest arose………</p>
<p> One of the important driving forces………</p>

</sec>
<sec numbering="2">
<title> A Perspective on Object-Oriented LW. Declaration Programming </title>
<p> A number of researchers argued that object …… </p>
<p> It has become customary to define the …… </p>

</sec>
<sec numbering="3">
<title> F-Logic by Example </title>
<p> The development of F-logic is guided by the desire to capture……… </p>
</sec>
<title> THE IS-A HIERARCHY. </title>
<p> Figure 2 shows part of a hierarchy of …… </p>
```
In the actual syntax of F-logic, we........ 

......
</sub_sec>

<sub_sec numbering="3.2"> 
<title> THE OBJECT BASE. </title>
<p> Figure 4 presents a database fragment .........</p>
<p> Note that F-logic does not ........</p>

......
</sub_sec>

......
</sec>

......

<reference> ABITEIKIUL, S., AND BEERI, C. On the .........</reference>

......

<reference> ZANIOLO, C., AiT-ffiCl, H., BEECH, D., CAMMARATA.......</reference>

</article>

<article id="2">
<title> Part-Whole Relationship Categories and Their Application in Object-Oriented Analysis </title>
<author> Renate Motschnig-Pitrik </author>
<author> Jens Kaasbøll </author>
<abstract> BPart decomposition and, conversely, the .....</abstract>
<iindex_term> Conceptual modeling, object-oriented ..... </iindex_term>
<sec numbering="1"> 
<title> INTRODUCTION </title>
<p> THE generalization/specialization .........</p>
<p> Due to recent and broad interest in modeling ........</p>

......
</sec>
<sec numbering="2"> 
<title> TERMINOLOGY AND PROPERTIES OF PART-WHOLE RELATIONSHIPS </title>
<sub_sec numbering="2.1"> 
<title> Terminology </title>


In order to ensure the proper 
An immediate consequence 

An immediate consequence 

Having identified composition
Relations belonging to the same

Having identified composition
Relations belonging to the same

[1] A. Artale, E. Francone, N.

[51] M.E. Winston, R. Chaffin,

Renate Motschnig-Pitrik
Jens Kaasbøll received

Extending SQL with Generalized Transitive Closure
Shaul Dar
Rakesh Agrawal
We present SQLiTC, an extension
Alpha-extended relational algebra

INTRODUCTION
THE ABILITY to process
We are interested in

MODEL
We first present the model...<p></p>

**Path Enumeration Operator PATHS:**

The PATHS operator......<p></p>

**Closure Predicate (λ):**

The λ predicate selects arcs from......<p></p>

The second type of conditions......<p></p>

......

......


[51] M. M. Zloof, "Query-by-......

Shaul Dar received the......

From 1989 to 1992 he was......

......

Rakeah Agrawal received......

From 1983 to 1989 he was......

......
Appendix C. XIL ALGORITHM

Class dataCollector retains the weighted result in a DOMTree Model from systems like TRIX. It has interfaces to mark a given element relevant based on the processed query.

class dataCollector{
    protected $domDoc;
    protected $rootNode;
    public function __construct() {
        $this->domDoc = new domDocument();
        $this->rootNode = $this->domDoc->documentElement;
    }

    public function getRootNodeElement(){
        return $this->rootNode;
    }

    /*
    * Function marks the node as relevant if the target
    * element exists in this q-tree, else the node is
    * irrelevant.
    */
    public function markElementRelevant($element){
        //Mark everything irrelevant first
        $this->markAllNodesIrrelevant($this->rootNode);

        $xpath = new DOMXPath($this->domDoc);
        $elements = $xpath->query($element);

        if (!is_null($elements)) {
            foreach ($elements as $element) {
                $element->setAttribute("relevant",RELEVANT_VALUE);
            }
        }
    }

    /*
    * This function marks every element in the node as
    * irrelevant used for structure based processing where
    * only the xpathquery
    * node name need to be set to relevant
    */
    public function markAllNodesIrrelevant($node){
        if ($node->hasAttributes()){  
            foreach ($node->attributes as $attribute){
                $node->setAttribute("relevant",IRRELEVANT_VALUE);
            }
        }

        if ($node->hasChildNodes()){
            foreach ($node->childNodes as $child)  
                $this->markAllNodesIrrelevant($child);
        }
    }
}

66
public function markRelevantForSelect(){
}

public function markRelevantForFrom(){
}

public function markRelevantForWhere(){
}

/*
 * This function prints correctly the attribute name
 * & value.
 */
public function print_attribute($node){
    if ($node->hasAttributes()){
        echo "<b>".$node->tagName."</b><br">".PHP_EOL;
        foreach ($node->attributes as $attribute){
            echo "AttributName: \".$attribute->name."\t";
            echo "AttributValue: ".$attribute->value."<br">".PHP_EOL;
        }
    }

    if ($node->hasChildNodes()){
        foreach ($node->childNodes as $child)
            $this->print_attribute($child);
    }
}

/*
 * This function will create a domDocument object which is
 * relevant at a given point of time.
 */
public function print_relevant($node){
    if ($node->hasAttributes()){
        echo "<b>".$node->tagName."</b><br">".PHP_EOL;
        foreach ($node->attributes as $attribute){
            if ($node->getAttribute("relevant")==RELEVANT_VALUE){
                echo "AttributName: \".$attribute->name."\t";
                echo "AttributValue: ".$attribute->value."<br">".PHP_EOL;
            }
        }
    }

    if ($node->hasChildNodes()){
        foreach ($node->childNodes as $child)
            $this->print_relevant($child);
    }
}

Class queryProcessor process the query for a given input.

class queryProcessor{
    protected $queryProcessed;
    protected $numQueryStrings;
    protected $queryStringArray;
protected $currentQueryProcessingIndex;

public function __construct($queryString) {
    $this->queryProcessed = $queryString;
    $this->queryStringArray = explode(" ", $queryString);
    $this->numQueryStrings = count($this->queryStringArray);
    $this->currentQueryProcessingIndex = $this->numQueryStrings - 1;
}

public function getCurrentQuerySubString() {
    $currQuerySubString = $this->queryStringArray[$this->currentQueryProcessingIndex];
    $this->currentQueryProcessingIndex = $this->currentQueryProcessingIndex - 1;
    return $currQuerySubString;
}

public function getNumQueryStrings() {
    return $this->numQueryStrings;
}

public function printQueryStringArray() {
    print_r($this->queryStringArray);
}

public function processStructureSyntax($dc, $subqueryString) {
    switch ($subqueryString) {
    case "select":
        $dc->markRelevantForSelect();
        break;
    case "from":
        $dc->markRelevantForFrom();
        break;
    case "where":
        $dc->markRelevantForWhere();
        break;
    default:
        break;
    }
}
}

Class XILEngine is main engine processing the query. It provides the main interface to an external client user interface (command line, web-based, or any graphical user interface).

class XILEngine{

    protected $dc;
    protected $inQuery;
    protected $qp;

    public function __construct($inputQuery) {

    }
/* Load the datacollection from the datacollector
 * In this prototype, it's assumed that the data collector
 * is a stats xml file of fixed size. */
 $this->dc = new dataCollector;
 $this->inQuery = $inputQuery;

 $this->qp = new queryProcessor($inputQuery);
}

public function processQuery(){
  /*
   * Get the subquery string element from query processor
   * and evaluate it.
   */
  for($i = $this->qp->getNumQueryStrings(); $i>0;$i--){
    $subqueryString=$this->qp->getCurrentQuerySubString();
    $this->processSubQuery($subqueryString);
  }
}

/*
 * This is for precessoring individual query item like
 * select, from and also e-clause elements.
 */
public function processSubQuery($subqueryString){
  /* First make decision on where it is a structure based
   * processeing for XML node names. If it is, they call
   * the markrelevant
   */

  global $xmlElementNames, $xilQuerySyntax;

  if (in_array($subqueryString, $xmlElementNames)) {
    $this->dc->markElementRelevant('//'.$subqueryString);
  }
  /*
   * Check the exact string from the query processor
   * and process based on the query word on the
   * DataCollector
   */
  elseif(in_array($subqueryString, $xilQuerySyntax)){
    $this->getQPObject()->processStructureSyntax($this->dc, $subqueryString);
  }
  else{
    echo "Not a query string: ".$subqueryString."<br">.PHP_EOL;;
  }
}

public function getInputQuery(){
  return $this->inQuery;
}

public function getQPObject(){
public function displayQtree()
{
    $this->dc->print_attribute
        ($this->dc->getRootNodeElement());
}

public function displayRelevantQtree()
{
    $this->dc->print_relevant($this->dc->getRootNodeElement());
}

public function printQPStringArray()
{
    $this->getQPObject()->printQueryStringArray();
}
Appendix D. TEXQUERY EXAMPLES (EXTENDED RESULT)

Query: //article[/p ftcontains 'logic' & 'programming' with stems window 2 without stopwords]/p

The result for the above query is as given in Table 19. The article, p element numbering and title enclosed in <! --- > are not part of the result but included here for reference.

Table 19 Results for TexQuery for a search on logic program

<table>
<thead>
<tr>
<th>Page Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;! --- Logical Foundations of Object-Oriented and Frame-Based Languages &gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;! --- 2 Perspective on Object-Oriented LW. Declaration Programming&gt;</td>
<td></td>
</tr>
<tr>
<td>A number of researchers argued that object-oriented languages are fundamentally different from (and even incompatible with) other paradigms, especially with <strong>logic programming</strong> [Maier 1989; Unman 1987]. This point of view was reflecting disappointment with the lack of early success in formalizing a number of central aspects.</td>
<td></td>
</tr>
<tr>
<td>&lt;! --- 3.2 THE OBJECT BASE.&gt;</td>
<td></td>
</tr>
<tr>
<td>Statement (vii) is a deductive rule that defines a new attribute, boss, for objects of class empl. It says that an employee’s boss is the manager of the department the employee works in. Here we follow the standard convention in <strong>logic programming</strong> that requires names of logical variables to begin with a capital letter.</td>
<td></td>
</tr>
<tr>
<td>&lt;! --- 3.3 Other features&gt;</td>
<td></td>
</tr>
<tr>
<td>.....</td>
<td></td>
</tr>
<tr>
<td>Arity-polymorphism is very popular in <strong>logic programming</strong> languages, such as Prolog.</td>
<td></td>
</tr>
<tr>
<td>.....</td>
<td></td>
</tr>
<tr>
<td>&lt;! --- 6 Predicates and Their Semantics&gt;</td>
<td></td>
</tr>
<tr>
<td>.....</td>
<td></td>
</tr>
<tr>
<td>To avoid confusion between icl-terms and P-molecules, we assume that predicates and function symbols belong to two disjoint sets of symbols, as in classical logic. In a practical <strong>logic programming</strong> language, however, this restriction may not be necessary and, as in classical logic programming theory, another useful purpose: Classical predicate calculus can now be viewed as a subset of F-logic. This has a very practical implication:</td>
<td></td>
</tr>
<tr>
<td>.....</td>
<td></td>
</tr>
<tr>
<td>Classical logic programming, too, can be thought of as a special case of <strong>programming in F-logic</strong>. With some additional effort, most of classical <strong>logic programming</strong> theory can be adapted to F-logic, which would make it upward compatible with the existing systems. Section 12 and Appendix A make a first step in this direction.</td>
<td></td>
</tr>
<tr>
<td>&lt;! --- 10. Herbrand’s Theorem&gt;</td>
<td></td>
</tr>
<tr>
<td>.....</td>
<td></td>
</tr>
<tr>
<td>Herbrand’s Theorem is a basis for the resolution-based proof theory in classical logic [Chang and Lee 1973]. In the next section we use Herbrand’s Theorem for F-logic to develop a similar result, which extends the results in Kifer and Wu [1993]. Just as classical proof theory was fundamental to the theory of <strong>logic programming</strong>, we</td>
<td></td>
</tr>
</tbody>
</table>
anticipate that the proof theory of F-logic will play a similar role in the object-oriented domain.

In this section, we define the notions of a logic program, a database, and a query, and illustrate the use of F-logic on a number of simple, yet nontrivial examples. We shall use the terms “deductive database” (or simply a database) and “logic program” interchangeably. As a first cut, we could say that a logic program in F-logic (abbr., an F-program) is an arbitrary set of F-formulas.

However, as in classical logic programming, this definition is much too general and both pragmatic and semantic considerations call for various restrictions on the form of the allowed formulas.

Perhaps the most popular class of logic programs is the class of Horn programs. A Horn F-program consists of Horn rules, which are statements of the form head + body, (8) where head is an F-molecule and body is a conjunction of F-molecules. Since (8) is a clause, this implies that all its variables are implicitly universally quantified.

Just as in classical theory of logic programs, it is easy to show that Horn programs have the model-intersection property. Thus, the intersection of all H-models of a Horn program, P, is aH-model of P. This model is also the least H-model of P with respect to set-inclusion (recall that H-structures are sets of molecules).

By analogy with classical theory of Horn logic programs (see, e.g., Lloyd [1987]), we can define an F-logic counterpart of the well-known TP operator that, given a Horn F-program, P, maps H-structures of P to other H-structures of P.

For generalized programs, the elegant connection between fixpoints, minimal models, and logical entailment holds no more. In fact, such programs may have several minimal models and the “right” choice is not always obvious. However, it is generally accepted that the semantics of a generalized logic program is given by the set of its canonic models, which is a subset of the set of all minimal models of the program.

Equality has always been a thorny issue in logic programming because it does not easily succumb to efficient treatment. As a result, most logic programming systems—and all commercial ones—have the so called “freeness axioms” built into them. ...

Clearly, programming in F-logic cannot avoid such problems. Furthermore, equality plays a much more prominent role in object-oriented programming than in classical logic programming.

F-programs specify what each method is supposed to do, define method signatures, and organize objects along class hierarchies.

These signatures are parametric; they declare the attributes first, rest, length, and the method append as polymorphic functions that can take arguments of different type. As with any clause in a logic program, variables in (11) are universally quantified.
To see how, suppose we want to verify that the result returned by one set-valued attribute stands in a certain relation (e.g., equals, contains, etc.) to the value of another attribute of some other object. Defining these relationships in F-logic is akin to comparing relations in classical logic programming:

On the other hand, while Definition 13.5 accommodates a vast class of programs, it is weaker than one may like it to be. As noted earlier, type-correctness in F-logic amounts to saying that well-typed programs are guaranteed to not imply facts that are inconsistent with signatures specified for these programs. Clearly, this is a minimum one should require of a correct logic program.

Because the above notion of well-typing is so general, it comes as no surprise that it is undecidable to check if an arbitrary program is well-typed [Kifer and Wu 1990/1991]. However, one can devise algorithms that would be sufficient for various special classes of programs. Such algorithms are a topic for further research; for classical logic programs, a similarly defined notion of type-correctness was studied in Wu [1992], both semantically and algorithmically. To give the reader a better idea about the envisioned use of the F-logic type system, we briefly mention the concept of @pe inference.

Originally, the notion of type-correctness was applied to strongly typed languages, languages that require the programmer to supply complete type specification with each program. However, in logic programming, complete type specification is often seen as a burdensome requirement.

While the idea of encapsulation is simple, its logical rendition is not. Miller [1979] suggested a way to represent modules in logic programming via the intuitionistic embedded implication.

LOGIN and LIFE [Ait-Kaci and Nasr 1986; Ait-Kaci and Podelski 1993] incorporate structural inheritance into logic programming via a unification algorithm for #-terms, which are complex structures related to signatures in F-logic but that are fundamentally different from the semantical point of view.

but in isolation from logic programming issues.

To integrate inheritance into an object-oriented logic programming system, such as F-logic, the following issues must be addressed:
This is analogous to classical logic, where logic programs with stratified negation have no complete proof theory.

In this paper, we proposed a novel logic that takes the works reported in Chen and Warren [1989] and Kifer and Wu [1989; 1993] to a new dimension: F-logic is capable of representing virtually all aspects of what is known as the object-oriented paradigm. We provided a formal semantics for the logic and showed that it naturally embodies the notions of complex objects, inheritance, methods, and types. F-logic has a sound and complete resolution-based proof procedure, which makes it also computationally attractive and renders it a suitable basis for developing a theory of object-oriented logic programming.

In classical logic programming, the equality problem can be side-stepped by banishing ‘<=’ from the rule heads.

Fortunately, there is a simple way out of these semantic quandaries (which also works for classical logic programs with equality).

In this appendix, we extend the results of Section 15.2.2 and Appendix A by defining perfect model semantics when negation and inheritance are working together. The Principle of Minimally Necessary Inheritance of Section 15.2.2 will continue to guide us here. However, it needs to be extended because semantics of general logic programs relies on making negative assumptions in essential ways.

A query optimizer needs to transform an SQL/TC query into an equivalent and more efficient form in which selections and projections are evaluated as early as possible. This problem has been studied extensively for logic programs involving more general recursion.

A large body of work has been done in recent years on the development of logic-based programming languages capable of expressing more general types of recursive queries, including non-linear recursion.