A Node Partitioning Strategy for Optimising the Performance of XML Queries

Gerard Marks, B.Sc., M.Sc.

A Dissertation submitted in fulfilment of the requirements for the award of Doctor of Philosophy (Ph.D.)

Dublin City University
Faculty of Engineering and Computing, School of Computing

Supervisor: Dr. Mark Roantree

August 2011
Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: _____________ ID No.: 56213109 Date: 15/08/2011
Acknowledgements

I would like to thank all those people who made this dissertation possible. In particular, I wish to express my sincere gratitude to my supervisor Dr. Mark Roantree for his time, patience, effort and excellent guidance throughout the PhD project. Without Mark, this dissertation would not have been possible.

I would also like to thank Dr. John Murphy for recommending me to Mark as a potential PhD candidate and for his support and advice on a number of occasions during the project. I may never have completed a PhD project if it were not for John’s initial encouragement.

Thanks also to Enterprise Ireland who supplied the funding for my research and to Dublin City University for the various structures and support they provided.

Thanks to my colleagues from the Interoperable Systems Group for sharing their experiences and knowledge during the time of my study. In particular, I would like to thank Martin and Jun for their help in thrashing out various ideas.

Thanks also to Paul Clarke and Donal McCann, for wading in through the December snow to do some last minute proof reading for me when everyone else was gone home for Christmas. It won’t be forgotten!

Thanks to my dad, Jim, and my sister, Caroline, for all their proof reading efforts and support, it was greatly appreciated.

A very special thanks goes to my fiancée Dervla for her love, devotion, and support throughout the entire PhD project. Not only was her emotional support abundant, but her valuable opinions and her proof reading, printing, and binding skills were all exploited.

This dissertation is dedicated to Dervla and my parents.
Contents

Abstract xi

1 Introduction 1
  1.1 The XML Data Model . . . . . . . . . . . . . . . . . . . . . . 3
  1.2 XML Query Processing . . . . . . . . . . . . . . . . . . . . 4
    1.2.1 Performance Issues in XPath Query Processing . . . . 6
  1.3 Aims and Objectives . . . . . . . . . . . . . . . . . . . . . . 8
  1.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2 Related Research 11
  2.1 Node Based Approaches . . . . . . . . . . . . . . . . . . . . 11
    2.1.1 The XPath Accelerator . . . . . . . . . . . . . . . . . 12
  2.2 Algorithm Based Solutions . . . . . . . . . . . . . . . . . . . 17
  2.3 XML Graph Indexing . . . . . . . . . . . . . . . . . . . . . . 18
    2.3.1 Strong DataGuides . . . . . . . . . . . . . . . . . . . . 19
  2.4 Path Based Approaches . . . . . . . . . . . . . . . . . . . . . 21
    2.4.1 Path Indexing Approaches . . . . . . . . . . . . . . . . 21
  2.5 Substituting Equijoins for Non-Equijoins . . . . . . . . . . . . 24
    2.5.1 XParent and the Ancestor/Leaf Index . . . . . . . . . 24
    2.5.2 Proxy Indexes . . . . . . . . . . . . . . . . . . . . . . . 26
  2.6 Node Partitioning Approaches . . . . . . . . . . . . . . . . . 27
  2.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3 The BranchClassIndex:  
   An Overview  
   3.1 Architectural Overview ..................................... 31  
   3.2 The City Bikes XML Repository ........................... 34  

4 XML Document Partitioning  
   4.1 Partitioning Constructs ................................. 38  
   4.2 The Initial Partition Set ................................ 39  
      4.2.1 Initial Partitions and False Hits ................. 41  
   4.3 Partition Refinement ..................................... 42  
   4.4 Query Processing ......................................... 45  
   4.5 Summary .................................................. 48  

5 Classification of Partitions  
   5.1 Branch Classification .................................... 50  
      5.1.1 Branch Classification ............................... 51  
      5.1.2 Typical Build Times and Storage Costs .......... 54  
      5.1.3 Worst Case Storage Costs for BranchClassIndex .. 56  
   5.2 Exploiting Branch Classification to Optimise XPath Queries  
      5.2.1 Modelling the Indexing Constructs ............... 59  
      5.2.2 Worked Example .................................... 60  
      5.2.3 Worked Example Summary ............................ 64  
   5.3 Extending the Branch Classification Process .......... 64  
      5.3.1 Identifying Text Values that have Low Selectivity .. 66  
      5.3.2 The Text Value Identification Algorithms ........ 68  
   5.4 Post Classification Integrity ........................... 72  
   5.5 Branch Classification Summary .......................... 74  

6 Query Processing  
   6.1 Index Deployment ....................................... 77  
   6.2 The Transformation Method .............................. 80
6.2.1 Generic Expression .......................................... 82
6.2.2 Transforming the Initial XPath Step ..................... 83
6.2.3 Transforming Interim Steps ................................. 85
6.2.4 Transforming the Final XPath Step ....................... 87
6.2.5 Transforming XPath Predicate Filters .................... 88
6.3 Sample Transformation ........................................ 90
6.3.1 Transforming the Generic Expression .................... 90
6.3.2 Transforming the Initial XPath Step ..................... 91
6.3.3 Transforming Interim Steps ................................. 91
6.3.4 Transforming the Final XPath Step ....................... 93
6.4 Index Selection .................................................. 96
6.4.1 Base Index Selection Rules ................................. 96
6.4.2 Advanced Index Selection Rules ......................... 97
6.5 Summary .......................................................... 99
6.5.1 Integrity Checking for Transformation Process ....... 99
6.5.2 Final Summary ............................................... 100

7 Experiments ....................................................... 101
7.1 Evaluation Method ............................................. 101
7.1.1 Implementation and Deployment Decisions ............. 103
7.1.2 Query Categories ........................................... 104
7.2 Specifying Low Selectivity Text Values .................... 104
7.3 The Bicycle Rental Dataset .................................. 105
7.3.1 Query Analysis .............................................. 106
7.3.2 Query Analysis after Text Value Classifications ...... 114
7.3.3 Overall Query Performance ............................... 119
7.4 Comparison Using Standard Benchmarks .................... 119
7.4.1 The XPathMark Benchmark ............................... 120
7.4.2 The Computer Science Bibliography .................... 123
7.4.3 The Protein Sequence Database ......................... 126
List of Figures

1.1 XML Document Illustrating XDM Properties ............... 3
1.2 XPath Twig Query Illustration .......................... 6

2.1 Pre/Post Encoding ....................................... 12
2.2 The XPath Accelerator Illustrated ......................... 13
2.3 Illustrating a Strong DataGuide .......................... 19
2.4 Relational Path Based Index .............................. 22
2.5 Comparing XParent and Ancestor/Leaf .................... 25
2.6 Partitioning factor $N=4$ ................................. 28

3.1 Indexing and Query Processing Architecture ............... 33
3.2 Single Station Sample for Lyon ........................... 36

4.1 Primary Partition Possibilities .......................... 41
4.2 Primary Partitions for Bicycle Rental Dataset ............ 42
4.3 After Splitting Large Partitions .......................... 45
4.4 Search Space Pruning using the BranchIndex ............... 46

5.1 XML Tree Showing Branch Classifications ............... 52
5.2 Full Binary Tree with Distinct Path for each Node ....... 55
5.3 ClassChain Components and Usage ......................... 58
5.4 Optimisation Constructs ................................ 59
5.5 Bicycle Rental Repository Subset ........................ 60
5.6 XML Tree Showing Branch Classifications ............... 65
5.7 Illustrating the Effect of Low Cardinality Text Values . . . . 66
5.8 Examining Text Value Operations . . . . . . . . . . . . . . 67
5.9 XML Tree Showing Branch Classifications . . . . . . . . . 73

6.1 XML Snippet from the Bicycle Rental Dataset . . . . . . . . 78
6.2 Node Relation and the NCLTV Index Relations . . . . . . . . 79
6.3 The XPath-to-SQL Transformation Template . . . . . . . . . 81
6.4 The Completed SQL Expression . . . . . . . . . . . . . . . . 95
6.5 Index Selection . . . . . . . . . . . . . . . . . . . . . . 96
6.6 Index Selection (Context Shift) . . . . . . . . . . . . . . . 98

7.1 Average Linear Path Expression Performance . . . . . . . . . 108
7.2 Average for Twig Queries without Text Values . . . . . . . . 109
7.3 Average for Twig Queries with Low Selectivity Text Values . 110
7.4 Average for Twig Queries with High Selectivity Text Values . 110
7.5 Average Single Step Path Fragment Queries . . . . . . . . . . 111
7.6 Average Performance across all Query Categories . . . . . . . 112
7.7 Average for Twig Queries with Low Selectivity Text Values . 115
7.8 Average Linear Path Expression Performance . . . . . . . . . 116
7.9 Average for Twig Queries without Text Values . . . . . . . . 118
7.10 Average for Twig Queries with High Selectivity Text Values . 118
7.11 Average Single Step Path Fragment Queries . . . . . . . . . . 119
7.12 Average Performance across all Query Categories . . . . . . . 120

8.1 Relational Index Deployment Revisited . . . . . . . . . . . . . 135
8.2 Redundancy Reduction . . . . . . . . . . . . . . . . . . . . . 136
## List of Tables

1.1 Breakdown of an XPath query. .......................... 5

2.1 XPath Expression Breakdown .......................... 14

3.1 Overview of the System Components .................. 34

3.2 Bicycle Rental Data Collection ........................ 35

5.1 Build Times ............................................. 54

5.2 BranchClassIndex Storage Costs ........................ 55

5.3 Extract from the Node Repository (Base Data) ........ 61

5.4 Extract from the NCLT Covering Index ................. 62

5.5 System Specification ..................................... 72

5.6 Efficiency of the Text Value Identification Algorithms .. 72

6.1 XPath 2.0 Language Coverage .......................... 80

6.2 The Generic Expression Components ................... 80

6.3 Populating the Class Option Variable .................. 82

6.4 The Initial Step Expression Components ................. 84

6.5 Populating the Class Option Variable .................. 85

6.6 The Interim Step Expression Components ................. 86

6.7 New Class Options ....................................... 86

6.8 Equality Options for Component \text{C12} ........... 87

6.9 Equality Options for Component \text{C13} ........... 87

6.10 Leaf Path Expression Components ...................... 88
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.11 Components within the <code>&lt;predicate-statement&gt;</code></td>
<td>89</td>
</tr>
<tr>
<td>6.12 Generic Expression Components</td>
<td>90</td>
</tr>
<tr>
<td>6.13 Initial Step Expression Components</td>
<td>91</td>
</tr>
<tr>
<td>6.14 First Interim Step Expression Components</td>
<td>92</td>
</tr>
<tr>
<td>6.15 The Second Interim Step Expression Components</td>
<td>93</td>
</tr>
<tr>
<td>6.16 Leaf Opening and Closing Components</td>
<td>94</td>
</tr>
<tr>
<td>7.1 Bicycle Rental Queries</td>
<td>106</td>
</tr>
<tr>
<td>7.2 Results for the Bicycle Rental Dataset</td>
<td>107</td>
</tr>
<tr>
<td>7.3 Branch Index Statistics</td>
<td>107</td>
</tr>
<tr>
<td>7.4 Bicycle Rental Dataset Results (after Text Value Classification)</td>
<td>113</td>
</tr>
<tr>
<td>7.5 Branch Index Statistics</td>
<td>115</td>
</tr>
<tr>
<td>7.6 XMark Queries</td>
<td>120</td>
</tr>
<tr>
<td>7.7 Results for the XMark Queries</td>
<td>121</td>
</tr>
<tr>
<td>7.8 BranchClassIndex Statistics (XMark)</td>
<td>122</td>
</tr>
<tr>
<td>7.9 Computer Science Bibliography Queries</td>
<td>124</td>
</tr>
<tr>
<td>7.10 Results for the Computer Science Bibliography</td>
<td>124</td>
</tr>
<tr>
<td>7.11 Branch Index Statistics (DBLP)</td>
<td>124</td>
</tr>
<tr>
<td>7.12 Protein Sequence Queries</td>
<td>126</td>
</tr>
<tr>
<td>7.13 Results for the Protein Sequence Database</td>
<td>127</td>
</tr>
<tr>
<td>7.14 Branch Index Statistics (Protein)</td>
<td>127</td>
</tr>
<tr>
<td>7.15 Results for the Node Based Approaches</td>
<td>129</td>
</tr>
</tbody>
</table>
Abstract

For ease of communication between heterogeneous systems, the eXtensible Markup Language (XML) has been widely adopted as a data storage format. However, XML query processing presents issues both in terms of query performance and updatability. Thus, many are choosing to shred XML data into relational databases in order to benefit from its mature technology. The problem with this approach is that (often complex and time consuming) data transformation processes are required to transform XML data to relational tables and vice versa. Additionally, many of the benefits of XML data can be lost during these processes. In this dissertation, we present a process that partitions nodes within an XML document into disjoint subsets. Briefly, as there are fewer partitions than there are nodes, a more efficient join operation can be performed between partitions, thus reducing the number of inefficient node comparisons. The number and size of partitions varies depending on the structure and layout in the XML document, and the number of partitions impacts query performance. Therefore, we also provide a partition classification process, which significantly reduces the number of partitions because each partition class represents many equivalent partitions within the XML document. In this dissertation, we will demonstrate that our approach outperforms similar approaches for a large subset of XML queries by eliminating complex join operations (where possible) during the query process.
Chapter 1

Introduction

A Markup Language can be used to annotate text with meaning. The Standard Generalised Markup Language (SGML) was adopted by the ISO in 1986 [6]. Contrary to what the name suggests, the SGML itself is not a markup language, but rather, a specification for defining markup languages. The best known application of SGML is the Hypertext Markup Language (HTML), which is used to annotate text in a way that web browsers understand.

The finite number of tags used in HTML soon became an issue because users wanted more control over web page rendering. Therefore, HTML was extended to include additional tags and fierce competition between Microsoft and Netscape fragmented the HTML standard. Hence, a new web page markup language was needed. At the time however, SMGL was considered too complex and therefore unsuitable for specifying the new web page markup language [6].

To overcome this issue, the eXtensible Markup Language (XML) was introduced in the late 1990’s. Similar to SGML, XML is a specification language for defining markup languages. However, contrary to SGML, XML is human readable. Therefore, the development of applications that process XML data is easier. One of the first applications of XML was XHTML (a reshaped
web page development language). However, XML generated wider interest because it provided a format in which any type of data could be stored and a common format in which heterogeneous systems could communicate. For these reasons, XML is today generally accepted as the de-facto standard for information interchange.

XML data is semi-structured, which means that each datum in an XML database has its individual structure attached. This is in contrast to structured (e.g. relational) databases, where a generic structure (i.e. a schema) must be designed first, and all of the data that one wishes to store, must conform to this structure. Making changes to this generic structure, for example to insert data that has an unsuitable structure, is often a time consuming task, and it can make applications that are dependant on the data function incorrectly.

In an XML database, heterogeneous data can be inserted seamlessly because each datum has its individual structure attached, and therefore does not have to conform to a global schema. This storage flexibility has resulted in systems generating large quantities of XML data. However, as the size of XML repositories grew, the tree-centric nature of XML data resulted in significant inefficiencies in terms of query performance - especially when compared to more structured database solutions.

Three recent case studies demonstrating XML's practical usage in industry were presented by the authors of [53]. Firstly, a Government Tax Agency was used to demonstrate that a relational database is impractical because of schema diversity (one or more tables would be required for each government form). As stated in [53], this would lead to thousands of relations in a relational database as well as issues in terms of schema evolution and selecting appropriate tables for join operations. Secondly, an Order Processing at a Telecommunications Company case study showed that the mapping of diverse orders to relational schemas is difficult and scatters the details of each
order across dozens of relations; it was noted that some orders were scattered across more than 100 relations. Finally, a study of an Event Logging at a Financial Services Company system showed that each event is variable and application dependent. Therefore, they cannot be easily mapped to a relational database schema.

The Sensor Web is another domain that is beginning to generate large quantities of data in XML format [61]. For example, in the domain of health and human performance, XML data is generated from sensors such as heart rate monitors worn by players in team sports [50].

1.1 The XML Data Model

XML itself is not a data model, but rather a specification for defining markup languages (as discussed earlier). However, in order to perform queries across XML data it is necessary to formally specify the individual properties of an XML document. For this purpose, the W3C recommend the XQuery and XPath Data Model (XDM) [68].

![Figure 1.1: XML Document Illustrating XDM Properties](image)

The work presented in this dissertation requires an understanding of four fundamental node types, which are specified in the XDM. An XML document contains a single root node called the document node (see Figure 1.1).
The document node contains an opening tag and a closing tag and all other nodes in an XML document will occur between these tags. The children of a document node must be element or text nodes. Other node types are permitted as children, for example comment nodes, but they are not relevant to this dissertation. A text node encapsulates XML character content. Similar to document nodes, element nodes have an opening and closing tag. However, there can be any number of element nodes in an XML document, whereas there is a single document node. Also, similar to a document node, an element node can have element and text nodes as its children. Unlike a document node, an element node can have one or more associated attribute nodes. Attribute nodes appear within an element node’s opening tag. An attribute node has a string-value, which is the normalised value of the attribute. In this dissertation, we refer to text nodes and the normalized value of attribute nodes collectively as text values.

1.2 XML Query Processing

The W3C recommend two query languages, XQuery and its fundamental subset XPath, as a standard means of retrieving data from an XML database. One of the most widely documented query performance issues is associated with hierarchical relationships [3,14,17,29,34,38,70], that is, the time it takes to resolve parent/child and ancestor/descendant relationships. In XPath, these relationships are specific to the ancestor, ancestor-or-self, descendant, descendant-or-self, parent and child axes, which we collectively refer to as the hierarchical XPath axes.

As XML database systems cannot perform at the same level as their structured counterparts, many of those who rely on XML for reasons of interoperability are choosing to store XML data in relational databases rather than its native format. The advantages of semi-structured data (e.g. schemaless data storage) are therefore lost in the structured world of relational
databases, where schema design is required before data storage is permitted. The result of this is that many domains, such as sensor networks, are using rigid data models where more flexible and dynamic solutions are required. Over the last decade, many research groups have developed new levels of optimisation. However, there remains significant scope and opportunity for further improvements.

To illustrate these problems, consider the following example.

**Example 1** *Retrieve the title of each masters thesis.*

**XPath:** `/descendant::mastersthesis/child::title`

<table>
<thead>
<tr>
<th>Step</th>
<th>Axis</th>
<th>NodeTest</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>descendant</td>
<td>mastersthesis</td>
</tr>
<tr>
<td>2</td>
<td>child</td>
<td>title</td>
</tr>
</tbody>
</table>

Table 1.1: Breakdown of an XPath query.

A *linear path expression* is an XPath query that does not contain predicate filters [28] (as illustrated in Example 1). Each linear XPath expression contains a number of *steps* (Table 1.1 shows the breakdown of the linear XPath expression in Example 1 into its steps). A step will take a sequence of nodes as input (the *context nodes*) and locate another sequence of nodes (the *target nodes*). The context node for the first step is the *document node*. The step’s *axis* specifies the relationship between the context and target nodes. For example, if the axis is *descendant*, then the target nodes must be descendants of the context nodes. In other words, target nodes must be in the subtrees rooted at the context nodes. A step will also contain a *NodeTest*, which specifies the *name* the target nodes must have and their *type*, for example *element* or *attribute*.

A *twig query* is an XPath expression that contains predicate filters [28]. In other words, a linear path expression locates a subtree within the target XML document and a twig query’s predicate filter(s) remove some of its
branches. A twig query is sometimes called a *tree pattern query* [5] because the query itself can be viewed as a tree (instances of which are located with the XML document).

For example, Figure 1.2a shows a sample twig query (the predicate filters are denoted by square brackets) and 1.2b illustrates its associated tree pattern. Within the tree pattern (graph), ancestor/descendant edges are denoted by double lines, whereas parent/child edges are shown as single lines. Figure 1.2c shows the tree representation an the XML document. The red edges show the paths that the twig query specifies and the thicker red edges are the path to the target node. The path marked with an ‘X’ are branches that failed to satisfy the predicate filter.

### 1.2.1 Performance Issues in XPath Query Processing

Sequence-oriented [30] evaluation of XPath steps is inefficient as all of the nodes in the sequence of context nodes must be compared, based on an XPath axis, to all of the nodes in the sequence of target nodes. Initially, the sequence of target nodes contains all of the nodes in the XML document.
(we refer to them as the initial target nodes). In a naive system, the entire sequence of target nodes will be traversed once for each context node or vice versa, which, as we will show, is inefficient.

Another issue with sequence-oriented evaluation of XPath steps is duplication of work. Duplication of work occurs when the ‘regions’ (of an XML document) ‘associated with the’ (XPath) ‘step are evaluated independently for each context node’ [32]; we refer to this as node-at-a-time evaluation of XPath steps. For example, based on the descendant axis, a single target node may be a descendant of multiple context nodes. If this occurs, a node-at-a-time processor will traverse the target node multiple times. As we will discuss shortly, node-at-a-time evaluation of XPath steps has a query performance overhead that is often unnecessarily large. The fundamental objective of an XML query processor is to reduce the number of nodes that must be visited during query processing.

Current XML query optimisation solutions can be placed in two broad categories. On one hand, index-based approaches build indexes on XML documents to provide efficient access to data. Index-based solutions, for example, XRel [69], XPath Accelerator [29], Xeek [42], benefit from existing join algorithms such as those that are available to standard Relational Database Management Systems (for example, NestedLoops, HashJoin, MergeJoin). Index-based approaches can also exploit mature relational facilities, such as Cost Based Optimisers, to select appropriate query execution plans based on specific XML data and queries. On the other hand, algorithm-based (or Native XML) approaches are focused on designing new join algorithms, for example, TJFast [38], StaircaseJoin [32], which are specifically designed to support queries across XML data.

The XPath Accelerator [29] is a node-at-a-time XPath query processor and it demonstrated that an XPath index stored inside a relational database can be used to evaluate all of the XPath axes. However, the XPath Accelerator
suffers from the aforementioned issues associated with node-at-a-time processors, such as duplication of work. Thus, the XPath Accelerator suffers from significant scalability issues as noted in [42].

Path-based approaches [24, 25, 34, 69] avoid visiting many nodes during the query process by storing each node’s root path [28] (the path from the document node to itself) in a path-index (sometimes called a path-summary index [7]). As many node instances can share the same root path there are usually much fewer root paths than there are nodes in the XML document. An XPath expression can be divided into multiple path fragments such as primary path fragments as specified in [24]. Node-at-a-time comparisons are only required between primary path fragments. Thus, if primary path fragments span more than one step in an XPath expression (which they often do), inefficient node-at-a-time comparisons are not required at every step in an XPath expression (unlike the node based approaches).

In a different approach, a special type of node partitioning allows nodes of different names and types (element/attribute) to reside in the same partition [43]. The motivation to do this is based on the fact that there will always be fewer partitions than there are nodes in the XML document. Thus, the partitions that contain the target nodes can be identified more efficiently and after the relevant partitions are identified, only the nodes that comprise these partitions need to be visited using costly node-at-a-time evaluations. However, to the best of our knowledge, [43] is the only such index-based node partitioning approach and it requires a user defined partitioning factor for each XML document. Thus, the user must run time-consuming experiments to identify a suitable partitioning factor for each XML document.

1.3 Aims and Objectives

The hypothesis put forward in this research is that larger numbers of node-at-a-time comparisons can be avoided during the query process through node
partitioning and partition classification. Furthermore, unlike the approach presented in [43], the time-consuming preprocessing stage that is used to identify suitable partitioning factors can be avoided. Finally, a node partitioning approach can be independent of particular XML node labels. This allows the most suitable XML encoding scheme to be chosen based on the user’s needs, while exploiting the performance benefits of the approach presented in this dissertation. To demonstrate the effectiveness of our approach, standard query performance benchmarks are used to compare this approach to that of other researchers and XML database vendor systems. In addition, data and queries, taken from a real world application which generates large quantities of sensor data in XML format are used to demonstrate the wider applicability of our approach. The main objectives of our research can be highlighted as follows:

- To provide a novel partitioning method for XML data storage that offers improved levels of optimisation for XML queries.

- To develop efficient algorithms that automatically identify and resize document partitions. This is unlike the existing approach that requires a preprocessing phase, which is infeasible for large XML documents (we present experiments to substantiate this claim).

- To exploit structural information to allow identical node partitions to be merged and thus, reduce the size of the index and avoid processing large numbers of equivalent node partitions during the query process.

- To provide a relational deployment of the indexing structures that encourages relational query optimisers to choose efficient query execution plans.

- To provide an XPath-to-SQL transformation process that produces SQL queries that are engineered specifically for the approach presented in this dissertation.
1.4 Summary

In this Chapter, a general introduction to XML and XML query processing was provided. XML has been widely adopted as a data storage format because XML data does not have to conform to a generic schema. In addition, XML databases can evolve easily without time-consuming schema re-design. XML also provides a common format in which heterogeneous systems can communicate, which has led to an explosive growth in its usage and the size of XML repositories as a whole.

A major obstacle to improving XML query performance is the tree-centric nature of the data and in this Chapter we discussed how node-at-a-time XPath step evaluation is not scalable. Thus, an XPath query optimiser’s primary objective is to visit as few nodes as possible during the query process. There are two main approaches to fulfilling this objective: (1) index-based approaches materialise data structures in advance of query processing to support query optimisation, (2) algorithm-based approaches bypass nodes by making decisions during query processing itself.

We begin in Chapter 2 by examining related solutions to XML query performance; in Chapter 3 an architectural overview of the system is provided and we introduce a real world XML case study; in Chapter 4, we provide a detailed description of the data structures used in the partitioned index and provide a step by step description of how it is built; in Chapter 5, we present the classification process for node partitions, which reduces the number of partitions while maintaining the same degree of search space pruning, and supports an additional node bypassing mechanism; in Chapter 6, we discuss index deployment and query processing; in Chapter 7, we present our experiments and discuss the findings; finally, Chapter 8 presents conclusions and future work.
Chapter 2

Related Research

There are several approaches to XPath query optimisation. In Chapter 1, we broadly categorised these efforts into index-based and algorithm-based approaches. Early index-based approaches to XPath query optimisation are presented in §2.1 and their inefficiencies are identified. Following this in §2.2, we provide an evaluation of how algorithm-based approaches overcome some of these shortcomings before introducing more advanced index-based approaches.

In §2.3, XML schema graph indexing is introduced as its concepts are used throughout the remaining index-based approaches; in §2.4, path-based indexes are evaluated; §2.5 discusses approaches that convert non-equijoins to more efficient equijoins to optimise node-at-a-time evaluations; finally, node partitioning approaches are discussed in §2.6.

2.1 Node Based Approaches

Node based approaches are those that do not use structure, (i.e. DataGuides, XML schemas, or Document Type Definitions), to optimise XPath steps. These approaches evaluate the XPath axes by comparing individual node labels in which the relationships between nodes are encoded.
2.1.1 The XPath Accelerator

The **XPath Accelerator** [29] is an XML index which is designed for deployment in relational databases. In this work, *pre/post* labels (known as **Dietz encoding** [22]), are used to encode each node with the region of the XML document that it encompasses. From the context of any given node, *pre/post* labels can be used to partition all other nodes in the XML document into its **ancestor**, **descendant**, **following** and **proceeding** nodes, that is, the four **major XPath axes**. Figure 2.1a depicts an XML tree that is labelled with (pre/post) identifiers and 2.1b illustrates how those nodes are dispersed in the pre/post plane. In particular, notice the four major XPath axes associated with node x (6,5). For example, the nodes that are ancestors of node x will have a preorder identifier that is less than 6 and a postorder identifier greater than 5. The other major axes can be resolved using similar pre/post logic.

![Figure 2.1: Pre/Post Encoding](image)

Figure 2.2 illustrates how a source XML document (2.2a) is stored in the node relation (2.2b). The first two columns in the node relation contain the **preorder** and **postorder** identifier of each node respectively. The major XPath axes can be sub-partitioned or extended (to include the context node itself) using the **minor XPath axes**. The most common of these are the **parent** and **child** axes, which sub-partition the ancestor and descendant axes.
respectively. For the purpose of evaluating the parent and child axes, the XPath Accelerator assigns an additional \( \text{par} \) (parent) label to each node. The \textit{yes} or \textit{no} (boolean) column \textit{att} is used for differentiating between \textit{attribute} and \textit{element} nodes as specified in the \textit{XQuery and XPath Data Model} (XDM) [68]. Node label \textit{tag} is the \textit{name} of the element or attribute node. Finally, \textit{cdata} is used to associate each node with its text value, i.e. text values associated with element or attribute nodes.

\textbf{Example 2 (Sample XPath Expression)}

\( //\text{article}[@\text{author} = 'T. Grust.']/\text{descendant::title} \).

The step-at-a-time XPath evaluation process that is used by the XPath Accelerator is referred to many times throughout this dissertation, thus we
Table 2.1: XPath Expression Breakdown

will now spend some time detailing it. The XPath expression in Example 2 specifies: *find the title of Torsten Grust’s articles*. The expression contains two steps and the breakdown of each step is shown in Table 2.1. Each step in an XPath expression receives a sequence of context nodes and locates another sequence of nodes, which we call the target nodes. The sequence of target nodes initially contains all of the nodes in the XML document. Then the sequence of initial target nodes is reduced to the actual target nodes based on the following:

- The step’s axis relative to the context nodes. For example, if the descendant axis is specified, the target nodes must be descendants of the context nodes.

- The node’s name (article) and type (element/attribute) to satisfy the step’s NodeTest.

- The text value of element or attribute nodes (T. Grust.).

The sequence of context nodes for the first step in an absolute XPath expression is always a single node sequence containing the document node. The document node is always assigned the first preorder and last postorder value, for example, node (0, 13) in Figure 2.2b. For all subsequent steps, the context nodes are the actual target nodes that were located at the previous step. Using the query in Example 2, the process is as follows:

1. Find the element nodes (we know it is an element node because attribute node names in an XPath expression have the prefix ‘@’) called article that are descendants of the document node. As all nodes are
descendant of the document node, in the first step all nodes that have 
tag\ ‘article’ and \att\ ‘no’ are returned from the node relation (Figure 2.2b), the sequence (1, 6, 11). For brevity, the sequence here contains the preorder identifiers only.

- The predicate filter, denoted by \textit{square brackets}, in the first step specifies that: \textit{only the articles that were written by Torsten Grust should be returned}. The first (and only) step in the predicate receives the sequence of context nodes: (1, 6, 11). The step’s axis is child, denoted by the abbreviated syntax ‘/’. The NodeTest specifies that the target nodes must have the \textit{tag ‘author’}, \textit{att ‘no’} and \textit{cdata ‘T. Grust.’} and they must be children of at least one of the context nodes. In this instance, the \textit{par} label is exploited to return those nodes that have the \textit{par} equal to 1, 6 or 11. In other words, all nodes whose parent is a context node are returned. The target node sequence is (12). Thus, as this predicate is a filter on the first step in the XPath expression, all of the target nodes in the first step that do not have node 12 as a child are filtered out. Therefore the sequence of target nodes is reduced to (11).

2. The final step receives the context node sequence (11) and the step’s axis is descendant. The NodeTest specifies that only nodes that have \textit{tag ‘title’} and \textit{att ‘no’} are returned. This time the preorder and postorder labels are used to ensure that the target nodes are descendants of node 11, that is, the nodes that have a preorder label greater than that of node 11, and a postorder label less than that of node 11. See \textit{pre/post} ranges in Figure 2.1 for more details. Thus, the node sequence (13) is returned.

In [31], a variant of the XPath Accelerator is presented, in that it uses
pre/size/level labels instead of pre/post/par. The level label replaces the par label used by the XPath Accelerator to evaluate the parent and child axes. The benefit of pre/size/level is that the size variant of post minimises the overhead of node relabelling upon updates to the XML document, as size is invariant with respect to subtree copying or moving, whereas post is not [64].

For the purpose of query performance however, [31] describes how partitioned B-trees (i.e. multi-column B-tree indexes) and context pruning can be used to optimise the performance of XPath queries in standard Relational Database Management Systems (RDBMSs). In this approach, partitioned B-trees are used to optimise non-recursive XPath axes (parent and child). Optimisation is achieved by minimising false hits; in other words, avoiding nodes that cannot contribute to the result. For example, a partitioned B-tree index on columns: (level, pre) allows the relational query optimiser to avoid false hits at levels that cannot contribute to the result. Additionally, as columns such as level and type have only a small number of possible values (that is, they have low selectivity), partitioned B-trees that are prefixed with these columns have lower creation and maintenance overheads than those prefixed on columns that have high selectivity [31].

Critical Evaluation

While the XPath Accelerator has the benefit of supporting all of the XPath axes, range comparisons (e.g. pre/post comparisons) were shown to be highly inefficient for large XML documents [40–42]. The experiments presented in this dissertation show that even when pushed to their limit using the techniques described in [31], node based approaches are inefficient because they perform too many node-at-a-time comparisons. The remaining approaches discussed in this Chapter describe ways of reducing the number of node comparisons that must be performed, therefore improving query performance.
2.2 Algorithm Based Solutions

The MPMGJN (Multi-Predicate MerGe JoiN) [70] algorithm is similar to the standard merge join algorithm used for equijoins in a relational database, but it is tailored to evaluate structural joins efficiently. A merge-join algorithm performs a join between two lists of nodes. Two cursors are created, one pointing to the head of each list. The cursors are compared to each other as they advance forward to perform the join operation. The MPMGJN differs to the standard merge-join by skipping nodes as the cursors are advanced forward [28].

In [3], it was noted that the MPMGJN could not evaluate the ‘/’ (child) axis efficiently in certain circumstances as it visits descendant nodes that are not actually child nodes. In [3], a StackTree is proposed which (unlike MPMGJN) avoids processing the unnecessary descendant nodes, thus improving performance. As the name suggests, StackTree uses a stack structure to store nodes that are nested on the same path in data trees. It was also shown in [63] that the MPMGJN and StackTree could be optimised further by calculating partitions of the pre/post plane that can be avoided, thus reducing the number of nodes that are evaluated.

The MPMGJN, StackTree, and similar approaches [32, 63], are binary join algorithms. A binary join in the case of XPath query processing is a join based on a sequence of context nodes and a sequence of initial target nodes at each step in the XPath expression. In other words, a binary join is a join between lists of nodes. This is exactly the approach used by the node based approaches detailed in §2.1. It soon became apparent that the binary join approach processed large numbers of intermediate nodes that could be avoided if more than two XPath steps are evaluated simultaneously.

PathStack and TwigStack use a more holistic approach to perform structural joins unlike the binary join algorithms. In contrast to StackTree, PathStack and TwigStack use multiple stacks to cache nested nodes. Each node in
a stack has a pointer to its corresponding node in its parent stack which enables the maintenance of possible \textit{n-ary} path solutions. Other advancements in holistic twig pattern matching solutions include TwigStackList [37], TwigList [59] and those that are based on prefix labelling schemes (much like Dewey decimal) TJFast [38], Twig2Stack [17].

\textbf{Critical Evaluation}

Relational database technology is a mature technology and thus, there are many advantages to indexing XML data within a relational database such as mature query optimisation and transaction management technology [30, 62]. In addition, modern query optimisers that are available in relational databases, such as the \textit{cost based optimiser} in Oracle 11g, are well suited for choosing suitable query execution plans for XML queries (if the XML index is well designed).

An issue with algorithm-based approaches (as identified in [31]) is that they require significant modifications to the relational database kernel (which is the approach used by [11]). Alternatively, \textit{Native} XML databases [23,33,51] can incorporate algorithm-based approaches to facilitate query optimisation.

\subsection{XML Graph Indexing}

Each datum in an XML database has its structure attached, which is the fundamental difference between semi-structured and structured data (structured data must conform to a global structure). Thus, structural information can be extracted from an XML dataset. In this section, we discuss various graph indexing schemes that exploit this structural information for the purpose of query processing.
2.3.1 Strong DataGuides

XML graph indexing is based on the concept of a Strong DataGuide [27].

Strong DataGuides are defined in terms of a graph that is not required to be a tree. When the graph is a tree, as is the case of an XML dataset, a Strong DataGuide reduces to what is known as a 1-index [28,52]. Therefore, we will now introduce the concept of Strong DataGuide in the form that is relevant to an XML tree, that is, as defined for a 1-index [52].

In a 1-index, a node in the source dataset (a data node) maps to a single distinct node in the index (an index node), that is, many nodes in the base data map to the same index node. One or more data nodes map to the same index node if they are B-bisimilar (backward bisimilar) [28]. Two data nodes are B-bisimilar if they have the same root path; the path that contains their name and the name of each of their ancestors in root-to-leaf order. Figure 2.3a shows a source dataset (taken from the Computer Science Bibliography [21]) containing data nodes. The 1-index is depicted in Figure 2.3b.

![Diagram](image)

Figure 2.3: Illustrating a Strong DataGuide
Critical Evaluation

There are two major issues with the 1-index. Firstly, the size of the 1-index can be equal to the size of the XML document in the worst case and even in situations where it is smaller than the size of the original dataset, it is usually too large to be efficient [28]. Similar approaches [16, 19, 36] have minimised the size of the 1-index by shortening the number of nodes in the root path. For example, by shortening the root path, the A(k) index [36] trades index size for query accuracy, i.e. the result set for the query may contain unwanted nodes, but the target nodes are definitely in the result set. However, more important is the fact the 1-index and similar approaches can only satisfy linear path queries, i.e. they cannot satisfy Twig queries [28].

It is widely thought that the smallest index that can satisfy all Twig queries is an F&B (Forward and Backward bisimilar) Index [1, 35, 65]. An F&B Index requires not only that the incoming root path is bisimilar, but also that the node’s outgoing paths be bisimilar. This is illustrated in Figure 2.3c. Notice how node 11 in the 1-index (Figure 2.3b) is separated from nodes 1 and 6 in the F&B Index. This is because node 11 has different outgoing paths than nodes 1 and 6. Similar is true for node 13.

Index graph approaches such as the 1-index cannot evaluate Twig queries, thus they provide coverage of a very small subset of the XPath language. Although an F&B Index can evaluate Twig queries, it is typically too large to be used in practice [28, 35, 65].

One approach [35] tried to minimise this problem by creating multiple F&B Indexes, each of which satisfies a subset of Twig queries. However, a more significant issue is that schema graph index structures are not suitable for indexing in a relational database [28, 39]. For this reason, (and because of the size of 1-indexes, F&B indexes and similar approaches), path-based indexing structures that prune the search space for Twig queries (rather than evaluating them) have become popular. Therefore, we will now discuss
2.4 Path Based Approaches

As discussed in §2.3, schema graph indexes are generally too large in practice. For this reason, many approaches exploit the same structural information as schema graph indexes, but for the simpler task of search space pruning. We have categorised this class of XPath index as path-based indexes.

The main similarity between path-based approaches and the graph indexing schemes (discussed in §2.3) is that the root path of each node is stored in a separate path index (sometimes referred to as path summary). Each distinct root path can only occur once in the path index and many nodes in the base data map to the same root path in the path index. Thus, regular path expressions can be executed across the path index to prune search space for linear path expressions. Additionally, Twig queries can often be evaluated as multiple linear path expressions.

2.4.1 Path Indexing Approaches

XRel [69] is a relational implementation of a path-based index. Figure 2.4 illustrates how a source XML dataset (2.4a) can be indexed in a relational database (2.4b and 2.4c). In XRel, each distinct root path (as described earlier for the 1-index) is assigned an identifier and is stored along with its path in a relation as shown in Figure 2.4c. Each node in the node relation (as shown in Figure 2.4b) is stored along with its path identifier. In other words, there is a many-to-one mapping from nodes in the node relation to their root path in the path relation.

Example 3 (Linear XPath Expression)

/\textit{dblp/article/title}.  

21
XRel divides a given XPath expression into one or more *simple path expressions* (similar to *regular expressions*). These simple path expressions can be executed across the path relation using the LIKE facility in the relational database. The LIKE keyword in SQL syntax allows regular path expressions to be executed across columns that contain character strings, such as the Path column in Figure 2.4c. Thus, the path identifiers for each simple path expression can be identified in the Path relation. After path identifiers have been determined, a *join* predicated on these path identifiers in the node relation will return all nodes associated with the simple path expression. Example 3 shows an XPath expression. In the Path relation, this XPath expression is mapped to the path identifier p4. All of the nodes in the node relation that have the path identifier p4 can be identified (nodes {3, 8, 13}).
The search space is pruned because those nodes that do not have the path identifier \( p_4 \) are avoided.

The authors of [24] show how a path-index (identical to that of XRel) can be exploited to evaluate multiple XPath steps in both forward axes (descendant, child) and backward axes (ancestor, parent) directions. This means that multiple contiguous descendant and child steps, which are called path fragments, can be evaluated simultaneously, as can contiguous ancestor and parent path fragments. In contrast, XRel can only evaluate path fragments that contain the child and descendant axes [24].

**Example 4 (XPath Twig Query)**

/\textit{dblp//title[.]/sub}.  

**Critical Evaluation**

There are a number of inefficiencies associated with path-based approaches. Firstly, if the XPath expression is a Twig query as in Example 4, the expression must be separated into multiple simple path expressions. In Example 4 there are two simple path expressions, these are: /\textit{dblp//title} and /\textit{dblp//title/sub}. In this instance, the path relation is used to determine the path identifiers associated with each of the path expressions in turn. The nodes associated with each of the path identifiers are located in the node relation resulting in two separate node sets. Inefficient node-at-a-time evaluations are then required between these two node sets. Earlier in this Chapter, we explained why this type of node evaluation is inefficient.

As discussed in §2.1, traditional node based approaches must perform this type of inefficient join once for each step in the XPath expression. Thus, path-based approaches provide query performance benefits over traditional node based approaches as there will be fewer joins when there are fewer simple paths than there are steps in the query expression. However, the experiments described in Chapter 7 show that large numbers of these node
comparisons are often required. In fact, we identify a category of XPath queries in which these approaches cannot provide any optimisation. According to the authors of [28], another inefficiency inherent in these approaches is that SQL can support exact string matching (across path indexes) efficiently using *equijoins* and B⁺-trees tree indexes on strings. However, they cannot efficiently support the regular expressions that are required to evaluate XPath’s *descendant* or *ancestor* axes. Additionally, [28] noted that regular path expressions associated with path-based approaches can produce incorrect results when recursion exists in the XML data. Reversed path approaches [18, 56] minimise these issues by reversing and then encoding the root paths associated with each node, but we believe that the much greater issue is the performance of inefficient join operations between path fragments (or their encoded alternatives).

### 2.5 Substituting Equijoins for Non-Equijoins

As discussed, the largest inefficiency associated with node based and path-based approaches is related node-at-a-time comparisons. These node comparisons are usually based on node ranges such as those in the pre/post plane. The authors of [40–42] pointed out that much of the inefficiency is a result of using non-equijoins to perform these range comparisons. In this section, a number of approaches are presented that substitute non-equijoins with more efficient equijoins where possible.

#### 2.5.1 XParent and the Ancestor/Leaf Index

To optimise XPath axes, XParent [34] proposed that the *transitive closures* of the XML tree can be precomputed and stored. In other words, one map from each node to its *set* of ancestor nodes and a second map to its *set* of descendant nodes are pre-materialised in the index. XParent uses a
a. Node Relation (Base Data)

<table>
<thead>
<tr>
<th>pre</th>
<th>post</th>
<th>par</th>
<th>att</th>
<th>tag</th>
<th>cdata</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13</td>
<td>null</td>
<td>no</td>
<td>dblp</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
<td>no</td>
<td>article</td>
<td>null</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>no</td>
<td>author</td>
<td>null</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>no</td>
<td>title</td>
<td>S. Manegold</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>no</td>
<td>sub</td>
<td>null</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
<td>no</td>
<td>i</td>
<td>null</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>0</td>
<td>no</td>
<td>article</td>
<td>null</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>6</td>
<td>no</td>
<td>author</td>
<td>null</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>6</td>
<td>no</td>
<td>title</td>
<td>null</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>8</td>
<td>no</td>
<td>sub</td>
<td>null</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>8</td>
<td>no</td>
<td>i</td>
<td>null</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>12</td>
<td>no</td>
<td>article</td>
<td>null</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>11</td>
<td>no</td>
<td>author</td>
<td>T. Grust.</td>
</tr>
<tr>
<td>13</td>
<td>11</td>
<td>11</td>
<td>no</td>
<td>title</td>
<td>null</td>
</tr>
</tbody>
</table>

b. XParent

c. Ancestor/Leaf

Figure 2.5: Comparing XParent and Ancestor/Leaf
relational database to deploy the index, therefore the transitive closures are stored in a relation as shown in Figure 2.5(b); in this relation, the left column contains ancestor nodes, and the right column contains their descendants. Query performance is gained because the ancestor and descendant XPath axes can be evaluated using equijoins across this relation, which are more efficient than non-equijoins [18,42] (range comparisons); range comparisons are used in the node-based approaches described earlier.

However, the size of the transitive closures is typically too large to be used in practice [28]. This is because many nodes in an XML document share the same ancestors and descendants, and thus, the XParent approach leads to a large amount of duplicated (redundant) storage.

The Ancestor/Leaf index [41] provides a more compact equijoin evaluation strategy. In this instance, each leaf node is stored along with its ancestors (in contrast to every node being stored with its ancestors and descendants) as depicted in Figure 2.5(c). This leads to less duplicated storage but since many leaf nodes share common ancestors, the storage costs of this approach are still significant.

**Critical Evaluation**

In practice, XParent and the Ancestor/Leaf indexes often incur prohibitive storage costs. However, it was shown that the Ancestor/Leaf outperforms the node-based approaches [40], which validates the use of equijoins over non-equijoins where possible. For this reason, the approach presented in this dissertation uses equijoins rather than non-equijoins for mapping between branch partition classes described later in Chapter 6.

**2.5.2 Proxy Indexes**

ProxyReach [41] focuses on reducing the duplicated storage resulting from the XParent and Ancestor/Leaf indexes. In this instance, a single proxy
node can be selected anywhere on the root-to-leaf path in an XML tree to represent a larger group of nodes. The proxy node is then stored along with all its ancestors and descendants. For example, if the proxy node is specified to be every leaf node, then the proxy index will be identical to the Ancestor/Leaf index. However, if proxy nodes are selected higher on the path, then the proxy can represent multiple root-to-leaf paths; thus, decreasing the storage requirements. Xeek [42] is a variant of ProxyReach in that it only requires that the ancestors of each proxy node are stored, thus further reducing the storage requirements.

**Critical Evaluation**

While proxy indexes have a benefit where storage minimisation is crucial, there is a trade-off between query performance and storage requirements. The reason for this is that many of the ancestors and descendants of proxy nodes represent false hits for the XPath step. In fact, the higher on the root-to-leaf path that proxy nodes are selected, the higher the number of false hits, that is, there will be fewer false hits in the Ancestor/Leaf index. These false hits must be removed using costly node comparisons such as those described for node based approaches earlier. In the case of these proxy indexes, *pre/post* labels are used (as with the XPath Accelerator). In summary, proxy indexes reduce storage requirements but to the detriment of query performance.

### 2.6 Node Partitioning Approaches

Node partitioning approaches segment documents into disjoint subsets. As there are fewer partitions than nodes in an XML dataset, a more efficient join operation can be performed between partitions, which reduces the workload for the more costly task of node comparisons. To the best of our knowledge,
[43] is the only major index-based approach in this area.

In [43], the pre/post plane is partitioned based on a user defined partitioning factor. Figure 2.6 illustrates the pre/post plane partitioned into parts using a partitioning factor of 4. For each node, the pre/post identifier of its part is the lower bound of its x and y values respectively. For example, in Figure 2.6, the part P associated with node x(6, 5) is P(4, 4). The ancestors of node x can only exist in the parts that have a lower bound x value \( \leq 4 \) and a lower bound y \( \geq 4 \), that is, the shaded parts (Figure 2.6). Similar is true for the other major XPath axes, for example, descendant, following and preceding.

**Critical Evaluation**

The problem with this approach is that an ideal partitioning factor is not known in advance and requires rigorous experimentation to identify. For example, in reported experiments each XML document was evaluated for the partitioning factors 1, 2, 4 up to 256 [43]. This type of experimentation is infeasible even for relatively small XML documents. Additionally, as XML data is often irregular, uniform partitioning of all nodes in an XML document based on a single partitioning factor may not be optimal. Finally, although it is suggested in [43] that the partitioning approach may
be tailored to other encoding schemes such as order/size, it relies heavily on the lower bound of each x and y value in the partitioned pre/post (or order/size) plane. Therefore, this approach does not naturally lend itself to prefix based encoding schemes such as [8, 10, 55], which have become popular in recent years because they facilitate updates more easily than region encoding approaches such as pre/post (discussed earlier in this Chapter).

2.7 Summary

In this Chapter, we have discussed related work in the area of XPath query optimisation. Initially, traditional node-at-a-time query evaluation strategies were discussed with their inefficiencies highlighted. We then discussed approaches that eliminate these inefficiencies. Native XML join algorithms (binary and n-ary) were then evaluated and their benefits discussed. We stated the reasons why these approaches cannot be deployed in a relational database without significant changes to its kernel, which suggests that an index-based approach that performs at the same level as their native counterparts may be a more preferable solution. For this reason, the remainder of the Chapter was dedicated to index-based solutions that can be deployed in a relational database.

We presented graph indexing schemes (for example, 1-index, F&B index) and described the subsets of the XPath language that they cover. We showed that these approaches often result in an index that is too large to be efficient. Also, it was stated that these approaches are not suitable for deployment in a relational database, but their concepts are used in path-based approaches which can be deployed in a relational database.

In terms of search space pruning, we described how path-based approaches provide significant performance gains over step-at-a-time processing of XPath expressions. However, these approaches often require inefficient node-at-a-time comparisons between path fragments, which are costly.
Between individual steps in an XPath expression or between path fragments, XParent and similar approaches show how inefficient non-equijoins can be converted to more efficient equijoins by explicitly storing the transitive closures of nodes. However, we described why the size of the transitive closures is often too large to be used in practice. Additionally, some approaches that reduce the size of the transitive closures were introduced, but it was shown that they are either still too large to be used in practice or they trade query performance for reduced storage costs.

Finally, we showed an alternative approach that uses disjoint node partitions. As there are fewer partitions than there are nodes, the partitions that contain the target nodes can be identified more efficiently, which reduces the number of costly node-at-a-time comparisons. However, this approach often requires significant preprocessing to identify suitable partitioning factors and it is heavily dependent on the properties of a single encoding scheme, which limits its possible application areas.
Chapter 3

The BranchClassIndex: An Overview

This chapter begins by introducing the architecture of the entire XML indexing and query processing system. The goal of this chapter is to provide a high level overview of the different steps in XML document processing and querying. Thus, it will provide the reader with a brief description of each of the system’s components.

While we benchmark our work against that of others using standard XML datasets and queries, we focus our attempts at query performance on a real world dataset to demonstrate the wider impact of our work. In §3.2, we provide an introduction to this real world XML dataset (the City Bikes XML repository).

3.1 Architectural Overview

In this section, a description of each of the processes is provided. Our objective is to familiarise the reader with each of these. We begin with an explanation of the overall system, which details the system’s components and the order in which each process and indexing construct is created and
executed, respectively.

The overall architecture is illustrated in Figure 3.1. Initially, an XML document \((d_1)\) is received by the partitioning process \((p_2)\). Thus, the partitioning process creates the index of node partitions: the \textit{branch index} \((i_3)\). Process \(p_2\) also identifies the properties of each node, such as \textit{name}, \textit{type} and \textit{level}, which serve as input to the process that creates the \textit{node repository} \((p_5)\). The node repository (sometimes referred to as the \textit{base data} \([35]\)) contains an entry for each node in the XML document. In other words, it is a complete representation of the original XML tree.

Due to the fine granularity of our partitioning process, the branch index will, for many XML documents, be too large. In other words, the branch index is exploited to prune the search space for queries, but searching the branch index provides an undesirable performance overhead. Therefore, we compact the branch index using a branch classification process \((p_4)\). The output of this classification process is the \textit{branch class index} \((i_9)\) in which a single branch class represents many branch instances - the classification process \((p_4)\) is the focus of Chapter 5. The class index is a compact version of the branch index.

Using the properties of each node that is received from the partitioning process \((p_2)\) and the branch class information received from the classification process \((p_4)\), process \(p_5\) generates the node repository \((i_6)\). Process \((p_7)\) subsequently generates the \textit{NCLT} (Name/Class/Level/Type) index \((i_8)\). As the node repository is deployed in a relational database, the \textit{NCLT} index can be generated using simple SQL expressions. Process \((p_7)\) generates the \textit{NCLT} index \((i_8)\) by selecting distinct \textit{name}, \textit{class}, \textit{level}, and \textit{type} from the node repository. Later, we will show how, based on the specific properties of our partitioning process, the \textit{NCLT} index can act as a \textit{covering index} \([35]\) for XPath steps that contain hierarchical axes. A covering index contains all of the attributes that are required by the query (or sub-query in this instance),
therefore the entire query can be evaluated in the index. In [31], they created a covering index for XPath steps by exploiting partitioned B-tree facility in a standard relational database [31] (as discussed in Chapter 2). In contrast, the NCLT index is a covering index that is specifically designed to optimise the hierarchical XPath axes and the improvement achieved using the NCLT index is substantial as we will show in Chapter 7.

Upon receiving an XPath query (q10), the XPath-to-SQL transformation process (p11) transforms the XPath query to its SQL equivalent (q12). The
SQL corresponding to each step in the XPath expression is an SQL sub-expression. Each of these sub-expressions in turn uses the class index \( i_9 \) to reduce the search space. The index selection process \( p_{14} \) then chooses when to query the high performance NCLT index \( i_{18} \) where possible, or else the node repository \( i_{16} \) will be selected. Finally, when each sub-clause is complete, the result set \( r_{15} \) is returned.

<table>
<thead>
<tr>
<th>Name</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning Process ( p_2 )</td>
<td>Chapter 4</td>
</tr>
<tr>
<td>Classification Process ( p_4 )</td>
<td>Chapter 5</td>
</tr>
<tr>
<td>Node Repository Creation Process ( p_5 )</td>
<td>Chapters 4 5</td>
</tr>
<tr>
<td>Generate the NCLT (Name/Class/Level/Type) Index ( p_7 )</td>
<td>Chapter 5</td>
</tr>
<tr>
<td>XPath-to-SQL Transformation Process ( p_{11} )</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>Branch Class Identification Process ( p_{13} )</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>Index Selection Process ( p_{14} )</td>
<td>Chapter 6</td>
</tr>
</tbody>
</table>

Table 3.1: Overview of the System Components

Table 3.1 provides a summary of the system’s processes and indicates the chapters in which more details can be found.

### 3.2 The City Bikes XML Repository

The use of sensors in the physical-world is constantly increasing and could now be regarded as widespread. The number of applications built on top of such sensor data is also increasing. Examples are urban traffic watch; weather monitoring; tracking of goods.

Recently the city of Dublin, like many other European cities, deployed a bike sharing scheme in which the public can rent (and return) a bike from stations located throughout the city centre. Stations are equipped with sensors that monitor bike availability and publish such data to the DublinBikes website (www.dublinbikes.ie). Users can connect to the website (through a PC or mobile application) to check where stations are, how many bikes are available, how many spaces are available to return bikes, and what type of
payment methods are available.

Using this data providers can understand at which station it is better to pick up or return bikes for maintenance in order to minimise service disruption. In effect, the web service offers an efficient mechanism for determining the current status of bike or space availability.

There are many situations in which it is advantages to be able to access historical data, or look for trends and patterns over time. For example, city planners or the companies offering the rental service must determine the location for new sites; determine those sites that require expansion; or reduce or close sites that are unpopular. Furthermore, this analysis must take place over time to avoid any bias that could result from poor weather patterns or other varying factors.

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
<th>Stations</th>
<th>Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aix-en-Provence</td>
<td>France</td>
<td>16</td>
<td>8 KB</td>
</tr>
<tr>
<td>Amiens</td>
<td>France</td>
<td>25</td>
<td>8 KB</td>
</tr>
<tr>
<td>Besancon</td>
<td>France</td>
<td>30</td>
<td>8 KB</td>
</tr>
<tr>
<td>Lyon</td>
<td>France</td>
<td>340</td>
<td>80 KB</td>
</tr>
<tr>
<td>Mulhouse</td>
<td>France</td>
<td>35</td>
<td>12 KB</td>
</tr>
<tr>
<td>Nancy</td>
<td>France</td>
<td>25</td>
<td>8 KB</td>
</tr>
<tr>
<td>Nantes</td>
<td>France</td>
<td>89</td>
<td>24 KB</td>
</tr>
<tr>
<td>Plaine-Commune</td>
<td>France</td>
<td>44</td>
<td>12 KB</td>
</tr>
<tr>
<td>Rouen</td>
<td>France</td>
<td>18</td>
<td>8 KB</td>
</tr>
<tr>
<td>Dublin</td>
<td>Ireland</td>
<td>40</td>
<td>12 KB</td>
</tr>
<tr>
<td>Toyama</td>
<td>Japan</td>
<td>16</td>
<td>8 KB</td>
</tr>
<tr>
<td>Luxembourg</td>
<td>Luxembourg</td>
<td>46</td>
<td>12 KB</td>
</tr>
<tr>
<td>Santander</td>
<td>Spain</td>
<td>13</td>
<td>4 KB</td>
</tr>
</tbody>
</table>

Table 3.2: Bicycle Rental Data Collection

The bicycle rental application [49] records information on bike availability in cities and towns across the world. The data is collected from each location at regular intervals (see Table 3.2) and the dataset at the time of our experiments (Chapter 7) was 2.06 GB in size.

A typical XML document for a single station in the city of Lyon (France) is shown in Figure 3.2. It shows that an entry was taken from Lyon on the 01/06/2010 at 19:59:50. The entry also illustrates the weather conditions and the unit of measurement. Finally, for each bicycle station, information
related to availability such as the number of free spaces and available bikes is provided. A segment of the bicycle rental dataset will be used to provide examples throughout the remainder of this dissertation.
Chapter 4

XML Document Partitioning

The motivation for document partitioning is that there will be fewer partitions in the XML document than there are nodes. Thus, the partitions that contain target nodes can be identified more efficiently; and the nodes that comprise all of the other partitions are eliminated from the search space. In §4.1, the new constructs that are used in the node partitioning process are introduced. This is followed in §4.2, with a step-by-step description of how the initial node partitions are created. The initial partitions are disjoint sets of nodes, which collectively contain every node in the XML document. However, we will show that the initial partitions act merely as a platform for XML optimisation and can be improved by avoiding unnecessary false hits [31]. Thus, the partitioning process is updated in §4.3 to create partitions of more desirable sizes with respect to the hierarchical XPath axes. In §4.4, a description of the query process for each of the hierarchical XPath axes is provided. Finally in §4.5, a summary of the concepts introduced in this chapter is provided.
4.1 Partitioning Constructs

This section introduces constructs that form part of the partitioning process. In [48], we defined a disjoint partition of nodes within an XML document as a *branch*. A branch construct and its sub-types are now described - illustrated examples of these constructs will follow in §4.2 and §4.3.

**Definition 1** A branch is a set of connected node identifiers within an XML document, where node identifiers are unique proxies for nodes in an index, for example pre/post labels.

A branch is the abstract data type used to describe a partition of nodes. In our work, we will deal with the local-branch and path-branch sub-types of a branch.

**Definition 2** A local-branch is a branch, such that its members represent a single branching node and the nodes in its subtree. A local-branch cannot contain a member that represents a descendant of another branching node.

The local-branch uses the branching node to form each partition. Our process uses the rule that each local-branch must not contain nodes that are descendants of another branching node to create primary partitions.

**Definition 3** A path-branch is a branch with a single path.

The path-branch is an abstract type with no branching node. Each member is a child member of the preceding node. Its three sub-types (*orphan-path*, *branchlink-path* and *leaf-path*) are used to partition the document.

**Definition 4** An orphan-path is a path-branch such that its members cannot belong to a local-branch.

The orphan-path definition implies that members of the orphan-path cannot have an ancestor that is a branching node. The motivation is to ensure that each node in the XML document is now a member of some partition.
Definition 5 A branchlink-path is a path-branch that contains a link to a single descendant partition of its local-branch.

In any local-branch, there is always a single branching node and a set of non-branching nodes. With the non-branching nodes, we must identify those that share descendant relationships with other partitions. These are referred to as branchlink-path partitions and each member occupies the path linking two branching nodes (i.e. two partitions).

Definition 6 A leaf-path is a path-branch that contains a leaf node inside its local-branch.

A leaf-path differs from a branchlink-path in that it does not contain a link to descendants partitions. In other words, it contains a single leaf node and its ancestors.

4.2 The Initial Partition Set

In the first attempt at partitioning, the goal is to include all nodes in local-branch or path-branch partitions. Path-branches are abstract types and at this point, all path-branch instances will be orphan-paths. Throughout this section, we describe the partitioning process with respect to element nodes.

In all cases, attribute and text nodes in an XML dataset are placed in the same branch as their associated element node. This results in fewer branches which can be exploited to boost query process.

The algorithms for encoding an XML document using a pre/post encoding scheme were provided by the authors of [29]. In brief, each time a starting tag is encountered a new object is instantiated with the following attributes of an element node: name, type, level, and preorder. Subsequently, the new element object is pushed onto a stack structure: the element stack. Each time an end tag is encountered an element is popped from the element stack and is assigned a postorder identifier.
Once an element has been popped from the stack, we call it the current node, and the waiting list is a set in which elements reside temporarily prior to being indexed. The first step in the process is to determine if the current node is a branching node by checking if it has more than one child node. The next steps are as follows:

1. If the current node is non-branching and does not reside at level 1 (one level greater than the level at which the document node resides), it is placed on the waiting list until step three (below).

2. If the current node is branching, it is assigned to the next local-branch in sequence. Also, the nodes on the waiting list that are its descendants are placed in the same local-branch and are removed from the waiting list.

3. If the current node is non-branching, but a node at level 1 is encountered, the current node does not have a branching node ancestor. Therefore, the current node is assigned to an orphan-path. For the same reason, any node currently on the waiting list is assigned to the same orphan-path.

At the end of this process, only the document node is unassigned. As the document node is a generic ancestor of all other nodes in the XML document, it can be ignored during the partitioning process. Indexing of the document node is described as part of the final index structure in Chapter 5. Figure 4.1 illustrates the set of local-branches LB-1 to LB-8 and orphan-paths OP-9 and OP-10.
4.2.1 Initial Partitions and False Hits

While Figure 4.1 shows the layout of branch instances in a hypothetical situation, we will now illustrate the layout of the primary partitions using a small segment from the real world bicycle rental dataset that was introduced in Chapter 3. The authors of [31] described false hits as the visiting of unnecessary nodes during the query process. In particular, they showed how false hits can be reduced by avoiding nodes at specific levels in the XML document that cannot contain target nodes. We will now illustrate the problem of false hits relative to the primary partitions and their impact on query performance before demonstrating (in §4.3) how a refinement of the primary partitions leads to fewer false hits, and thus, improves query performance.
Figure 4.2 illustrates the primary partitions for a small segment of the bicycle rental dataset. In this instance, there are four local-branches and no orphan-paths. To illustrate the concept of false hits, we will analyse the query process for the ancestor axis. Given a sequence of context nodes, we must identify the branch instances that the context nodes belong to, and their ancestor branches. If the sequence of context nodes contains nodes 11 and 12 (Figure 4.2), local-branches LB-2 and LB-4 will be identified. The search space is pruned at this point because nodes 5, 6, 7, 8, 18, 19, 20 and 21 do not reside in LB-2 or LB-4 and thus, are not visited during query processing. However, there are only three nodes within local-branch LB-4 that can be hierarchically related to context nodes from LB-4, these are nodes 1, 9 and 10. This means that nodes 2, 3, 4, 15, 16, and 17 are guaranteed false hits. False hits can be eliminated using individual node comparisons (e.g. using pre/post labels), but this is inefficient (as discussed in Chapter 2).

Figure 4.2: Primary Partitions for Bicycle Rental Dataset

4.3 Partition Refinement

We have identified a principle that minimises the number of false hits: only hierarchically related nodes are permitted in each branch and local-branches can contain nodes that do not have a hierarchical association (i.e. an ances-
tor/descendant relationship). When nodes within branches do not share a hierarchical association, the false hits previously described will occur (causing inefficiencies for the hierarchical XPath axes).

Each local-branch instance has a single branching node root which may have many (non-branching node) descendants. It is the non-branching descendants of the root that are examined to determine if they share a hierarchical association as we want to ensure that only nodes that are hierarchically related reside in the same partition. Thus, we partition the non-branching nodes (in each local-branch) into disjoint path-branches (Definition 3). As orphan-paths and local-branches are disjoint, each of these path-branch instances will be a branchlink-path or a leaf-path.

The RefinePartitions (Algorithm 1) replaces all steps outlined for creating the primary index (above). The new branch partitions are created by processing two local-branches simultaneously. All current nodes (see creating primary partitions above) up to and including the first branching node are placed in the first waiting list (wList1) where they wait to be indexed. Subsequently, the next set of current nodes, up to and including the second branching node, are placed on the second waiting list (wList2). At this point, wList1 and wList2 contain the nodes that comprise the first and second local-branches respectively.

If a node at level 1 is encountered, the nodes that comprise wList2 are an orphan-path (line 2). If a branchlink-path exists, RefinePartitions identifies it as the non-branching nodes in wList2 that are ancestors of the root node in wList1 (lines 6-7). If one or more leaf-paths exist, they will be the nodes in wList2 that are not ancestors of root node in wList1 (lines 8-9). The remaining nodes that comprise the first local-branch (wList1) are then moved to the index (line 12); this will be the single branching node root of the first local-branch only. At this point, the only node that remains in wList2 is the root node of the second local-branch. This local-branch
Algorithm 1 RefinePartitions

1: if node at level 1 encountered then
2:   move nodes that comprise wList2 to orphan-path;
3: end if
4: move non-branching nodes from wList1 to leaf-path;
5: for each node n in wList2 do
6:   if n = ancestor of wList1.ROOT \& n ≠ branching node then
7:     move n to branchlink-path;
8:   else if n ≠ ancestor wList1.ROOT then
9:     move n to leaf-path;
10: end if
11: end for
12: move local-branch from wList1 to local-branch;
13: move local-branch from wList2 to wList1;

is then moved to wList1 (line 13) and thus, wList2 is emptied. The next local-branch is placed in wList2 and the process is repeated until all nodes are processed. When this process has completed, the result will be many more partitions, with the benefit of increased pruning.

A second function of this process is to track and index the ancestor-descendant relationships between branch partitions. This is achieved by maintaining the parent-child mappings between branches. Given two branch instances: B1 and B2, B2 is a child of B1 if and only if the parent node of a node that is in B2 belongs to B1. When the RefinePartitions process is complete, the ancestor-descendant relationships between branches are determined using a recursive function across these parent-child relationships, that is, by selecting branch’s children, then its children’s children recursively.

The layout of the final branch possibilities is illustrated in Figure 4.3. In particular, notice that the nodes within each branch instance are hierarchically related and branching nodes always end up in a single node branch, which is important for the query process, which we describe next.
4.4 Query Processing

The branch index (which we refer to as BranchIndex) has some very useful properties which can be exploited for the purpose of query performance. This is illustrated in Figure 4.4. Notice how the BranchIndex is exploited to prune the search space by limiting the region of the node repository that must be evaluated. Moreover, these properties are fundamental to the approach described in the subsequent sections and are thus explained here.

Consider the six hierarchical XPath axes: ancestor, ancestor-or-self, descendant, descendant-or-self, parent and child. We described in Chapter 2 how a step in an XPath expression receives a sequence of context nodes. Then, from an initial sequence of target nodes (the entire XML document), the
sequence of actual target nodes are located by performing a join between the two sequences. This join operation is based on the step’s axis, NodeTest and predicates. In our approach, we achieve performance improvements by firstly (before the join operation) identifying the set of branch instances associated with the sequence of context nodes. The branch instances that are identified are based on the step’s axis as follows:

- **ancestor, ancestor-or-self axes.** Identify the ancestor-or-self branches associated with each context node.

- **descendant, descendant-or-self axes.** Identify the descendant-or-self branches associated with each context node.

- **parent axis.** Identify the parent-or-self branches associated with each context node.

- **child axis.** Identify the child-or-self branches associated with each context node.
In each case, the self branch is the branch in which the context node resides and the remaining branches are those that are hierarchically related based on the step’s axis. Thus, the identified branches will always contain the target nodes. It is important to note that, after the identification of the branch instances associated with the sequence of context nodes, a node’s level attribute is the only additional attribute required to determine the remaining hierarchical information for the step. Thus, the following attributes can be used to create a covering index for an entire XPath step:

**Example 5 (covering index attributes)**

*name, \{branches\}, level, type.*

These attributes are different from traditional approaches as node labels such as pre/post or Dewey decimal [55, 62] are not required to satisfy the hierarchical relationships between nodes. If they were required, the covering index would be at least as large as the original XML document as pre/post or Dewey decimal labels are always unique for each node. To exemplify how name, \{branches\}, level and type are exploited to satisfy hierarchical relationships, consider the single context node sequence containing node 8 (Figure 4.3). If the step contains the descendant axis, the query process is as follows:

1. Identify the descendant-or-self branches: LP-4, LP-5, LP-6, LP-7, LP-8, LB-9, LP-10, LB-11, BLP-12, LB-13.

2. Check the name, level and type of each node in these branches to determine the sequence of target nodes.

At this point, there are still too many branch instances for this system to be very efficient for most purposes. This provides the motivation for reducing the size of the BranchIndex in Chapter 5. After the size of the BranchIndex is reduced, we will show how the *name, \{branches\}, level,*
type attributes become the basis of a highly optimised covering index for XPath steps.

4.5 Summary

In this chapter, we introduced the constructs that are used in the partitioning process (a branch and its sub-types). The creation of primary partitions demonstrated that partitions can be created based on the occurrence of branching nodes within an XML document, which avoids preprocessing to identify suitable partitioning factors; an issue we identified for the approach most similar to ours [43] (see Chapter 2). We then showed how an index of primary partitions can result in a large number of false hits for the hierarchical XPath axes (as nodes within an individual branch partition may not share a hierarchical association), and discussed their associated performance overhead. To address the issue of false hits, we discussed how partitions can be resized. For this purpose, a new strategy that uses the RefinePartitions algorithm was presented, which creates branch partitions using the rule that nodes within a branch must be hierarchically related.
Chapter 5

Classification of Partitions

The BranchIndex, resulting from the branch partitioning approach presented in Chapter 4, provides a search space pruning method for XPath steps. However, as we pointed out, this technique has a large index storage cost because the fine granularity of the partitioning process results in a large number of branch instances. Thus, traversals of the BranchIndex have a performance overhead that reduces the gains achieved through search space pruning. In this chapter, we describe a technique which reduces the size of the BranchIndex using a branch classification process. The BranchClassIndex, which is a specialised version of the BranchIndex, maintains all of the benefits for a reduction of the storage and therefore index traversal costs. In addition to search space pruning, (as an additional benefit of the branch classification process), we will introduce our node bypassing strategy for XPath steps.

This chapter is structured as follows: in §5.1, the branch classification process is introduced; §5.2 describes how the BranchClassIndex is exploited to optimise XPath queries; in §5.3, we describe how the classification process can be extended to achieve further query performance gains; finally in §5.4, we conclude by demonstrating that the integrity of result sets is maintained after the classification process.
5.1 Branch Classification

The final phase in constructing the index is to reduce its size while maintaining the same degree of search space pruning - this is process p4 in the system architecture (Classification Process). To achieve this, branch instances are classified into *branch classes* which are used to construct the final index (the BranchClassIndex). In the BranchClassIndex, a single branch class represents many branch instances.

Before presenting the classification process, we will discuss how the properties of the BranchIndex migrate to its specialised version (the BranchClassIndex). Recall from Chapter 4 that based on the special properties of the branch partitioning process and given a sequence of context nodes, the hierarchical XPath axes can be satisfied using the following attributes of a node:

**Example 6 (original covering index attributes)**

*name*, \{branches\}, *level*, *type*.

In the same chapter, we also demonstrated how these properties can be exploited to create a covering index for steps in an XPath expression that contains a hierarchical XPath axis. The problem that was noted however, is that there will usually be too many branch instances in an XML document to realise significant performance gains. In contrast, now based on the classification of branch instances, an optimised covering index for hierarchical axes can be achieved using the following attributes of a node:

**Example 7 (new covering index attributes)**

*name*, \{branch classes\}, *level*, *type*.

We refer to the index that is based on these attributes as the NCLT (Name, Class, Level, Type) index. In the NCLT index, a single *branch class proxy* represents many branch instances. Therefore, we will show how large numbers
of nodes can be bypassed during the query process. The overall optimisation strategy can now be viewed in two phases:

1. For hierarchical XPath steps that must access the base data, the search space is pruned by exploiting the branch information that is implicit in the branch classes that are indexed.

2. For instances in which the NCLT covering index can be exploited to optimise an XPath step, large numbers of nodes can be bypassed because a single branch class proxy can be evaluated in place of a large number of branch instances. This chapter will show that, when the NCLT index is used, search space pruning and node bypassing are achieved simultaneously.

**Definition 7 (Generic Ancestor)**

A generic ancestor is a node that is an ancestor of all nodes in an XML document with the exception of the root and other generic ancestor nodes.

For example, in Figure 5.1 (below), node 1 is a generic ancestor because it is an ancestor of all nodes except the root node. As will be shown shortly, the root node, and generic ancestor nodes, can be ignored during the branch classification. Therefore nodes can be indexed earlier (removed from main memory) making the classification process more efficient.

### 5.1.1 Branch Classification

The branch classification process will place every branch instance in a single branch class. To achieve this, each XML document is treated as a set of disjoint sub-documents where each sub-document is rooted at the first level that contains a non-generic ancestor. In Figure 5.1, the root node resides at level 0. The only generic ancestor in this example resides at level 1. Thus, sub-documents 1-4 are rooted at level 2.
The classification process we are about to present is in some ways similar to the concept of forward and backward bisimulation [1, 35]. The first major difference is that we are classifying branch instances rather than node instances. Secondly, paths leading to element and attribute node names are considered in the branch classification process, but those leading to their text values are not. For example, in Figure 5.1, the text values (e.g. $iValue_1$, $kValue_1$) are not included in the classification process, therefore they do not reside inside a sub-document.

**Example 8 (Forward Paths of Node 3)**

\[
/c/d/e/\{\}, \ c/d/f/\{/g, /g/h, /g/h/i, /g/h/j\}, \ c/d/f/g/\{/h, /h/i, /h/j\}, \ c/d/f/g/h/\{/i, /j\}, \ c/d/f/g/h/i/\{\}, \ c/d/f/g/h/j/\{\}
\]

The steps in the branch classification process for each sub-document are as follows:

**Step 1: Calculate Forward Path Identifiers.** Traverse each element node in the sub-document and assign to it a forward path identifier. We calculate this identifier for each element node using the dash separated pair $[\text{root path}-\{\text{ordered outgoing paths}\}]$. For instance, the forward path for node
Recall from Chapter 4 (XML Document Partitioning) that the partitioning process exploits a stack structure to process element nodes. In short, when an opening tag is encountered the new element is pushed onto an element stack. When a closing tag is encountered, the element on the top of the element stack is popped (removed from the stack) because the end of the node has been reached. Using the same stack based approach, the steps in calculating the forward bisimulation are as follows:

1. When an opening tag is encountered, add the root path (the path containing the names of the nodes from the root of the sub-document to this node) of this new node to the set of forward paths of each node currently on the element stack.

2. When a closing tag is encountered:
   - The top element is popped from the stack, and its [root path-\{outgoing paths\}] pair (described above) is added to the set of forward paths that each node on the element stack maintains.
   - Assign a unique forward identifier to the forward paths of the node just popped from the stack, or, get the previously assigned identifier if the same path was encountered before.

This process has worst case O(N*M) complexity, where N is the number of nodes in the sub-document and M is the maximum depth of the tree structure (the maximum possible number of nodes on the element stack).

Theoretically, M can be large, but it is common for the maximum depth of an XML document to be less than 10 levels, for example DBLP [21] (6 levels), Protein Sequence Database [58] (7 levels). In fact, the XPathMark benchmark [67] for stress testing the performance of XPath queries against synthetic XML data uses just 13 levels.
**Step 2: Calculate Backward Path Identifiers.** Traverse each element node in the sub-document one more time and assign to it, a backward path identifier. A node’s backward path identifier is calculated using its *root path’, where the *root path’ is the path of containing the forward path identifier of each node (calculated in the previous step) from the root node, within the sub-document, to this node.

As with forward path identification, this process has worst case O(N*M) complexity because the element stack is traversed for each node in the sub-document to find the *root path’ of the node.

**Step 3: Classify Branch Instances.** For each branch, its branch class identifier is calculated as backward path identifier of each element node in the branch, in document order.

### 5.1.2 Typical Build Times and Storage Costs.

The build times for the four XML datasets evaluated in this dissertation are shown in Table 5.1 - these datasets are formally introduced in Chapter 7. The build times for the same datasets using SQL Server 2008 and MonetDB/XQuery are also shown for comparison purposes. MonetDB/XQuery took the least amount of time across all datasets, followed by the Branch-ClassIndex and SQL Server, respectively. SQL Server has the least efficient indexing process across all of the XML datasets. Also, for SQL Server, a build time is not shown for the Computer Science Bibliography (DBLP) because it had difficulty processing and XML document with an associated DTD (Document Type Definition).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>BCI</th>
<th>SQLS</th>
<th>MonetDB</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>676 MB</td>
<td>4.70</td>
<td>——</td>
<td>1.56</td>
<td>Minutes</td>
</tr>
<tr>
<td>XMark</td>
<td>1.33 GB</td>
<td>9.93</td>
<td>36.73</td>
<td>3.34</td>
<td>Minutes</td>
</tr>
<tr>
<td>Protein</td>
<td>683 MB</td>
<td>7.86</td>
<td>37.28</td>
<td>1.69</td>
<td>Minutes</td>
</tr>
<tr>
<td>Bikes</td>
<td>2.06 GB</td>
<td>22.55</td>
<td>146.62</td>
<td>18.30</td>
<td>Minutes</td>
</tr>
</tbody>
</table>

Table 5.1: Build Times
The BranchClassIndex took more time than MonetDB/XQuery to build the index; the difference for each dataset in minutes is: 3.14 (DBLP), 6.59 (XMark), 6.17 (Protein), 4.25 (Bikes). In other words, for the four datasets respectively, MonetDB/XQuery is 100.30%, 99.32%, 129.21%, 20.8% more efficient at building the index. However, MonetDB requires ‘modifications to the internals of the underlying RDBMS kernel’, whereas the BranchClassIndex can be deployed in an ‘Off-the-Shelf’ relational database [31].

<table>
<thead>
<tr>
<th>Dataset</th>
<th>XML Doc.</th>
<th>BaseData</th>
<th>BCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>676 MB</td>
<td>951 MB</td>
<td>269.40 KB</td>
</tr>
<tr>
<td>XMark</td>
<td>1.33 GB</td>
<td>1.22 GB</td>
<td>117.23 MB</td>
</tr>
<tr>
<td>Protein</td>
<td>683 MB</td>
<td>853 MB</td>
<td>149.76 MB</td>
</tr>
<tr>
<td>Bikes</td>
<td>2.06 GB</td>
<td>2.56 GB</td>
<td>80.04 KB</td>
</tr>
</tbody>
</table>

Table 5.2: BranchClassIndex Storage Costs

The layout of the BranchClassIndex is detailed in Chapter 6. Table 5.2 shows the storage cost (size on disk) of the BranchClassIndex. The BaseData is the representation of the original XML dataset within Oracle 11g relational database and BCI is the storage costs of the BranchClassIndex. For each XML dataset, the BaseData is larger than the original XML document. This is because it contains the encoded (pre/post) structure of the XML document and indexing attributes associated with the BranchClassIndex, such as the branch class identifier of each node, are stored in the BaseData. Crucially however, for each dataset the storage costs of BranchClassIndex is smaller than the BaseData, for example, 269.40 kilobytes (DBLP), 117.23 megabytes (XMark).

Figure 5.2: Full Binary Tree with Distinct Path for each Node
5.1.3 Worst Case Storage Costs for BranchClassIndex

Recall, from Chapter 4, that the BranchIndex contains the ancestor/descendant and parent/child relationships between branch instances. Therefore, because the BranchClassIndex is a compact version of the BranchIndex it must contain the ancestor/descendant and parent/child relationships between branch classes.

In the worst case, there will be as many branch instances as there are nodes in the XML document. For instance, a full binary tree (sometimes called a proper binary tree) is a tree in which each non-leaf node has exactly two child nodes. According to our definition of branch instances, branching nodes always reside in a single node partition. Thus, if there are no attribute nodes in the XML tree (because they reside in the same branch as their parent element) and it is a full binary tree, each branch instance will contain exactly one node (a single branching node or a single leaf node). Furthermore, if each node in this XML tree has a distinct root path (that is, it does not share its root path with any other node in the tree), there will be as many branch classes as there are nodes in the XML document - such an XML tree is shown in Figure 5.2.

To index the relationships between branch classes in a relational database, we adopted the approach similar to that proposed by XParent [34] (see Chapter 6 for details). In XParent, the relationships between nodes - or the transitive closures of nodes - are stored explicitly in an Ancestor table. This ancestor table allows XPath steps to be answered using equi-joins rather than \( \theta \)-joins (nonequi-joins). Thus, if there are as many branch classes as there are nodes in the XML document, the performance of the BranchClassIndex reduces to that of XParent.

As our approach in its worst case can, like XParent, use equi-joins rather than \( \theta \)-joins, hash-join algorithms are an option of the query optimiser. A Hash join has \( O(N+M) \) complexity. However, the inner row of a hash-join
must be loaded into memory. Thus, in our worst case scenario, the query optimiser may choose nested-loops, which has a worst case complexity of \(O(N \times M)\).

5.2 Exploiting Branch Classification to Optimise XPath Queries

In Chapter 4, we described how branch instances are exploited to prune the search space during query processing. In particular, a description was provided of how the branch instances (that prune the search space) are identified for each XPath step based on its axis and sequence of context nodes. We now show how this process is optimised using the BranchClassIndex. For this purpose, we introduce the concept of a ClassChain. This is followed by a description of how ClassChains are exploited to improve the performance of XPath steps. The section is completed with a worked example which illustrates how the query process associated with the BranchClassIndex differs from that used by other approaches.

**Definition 8 (ClassChain)**

For any branch class (which we will call the self-class), a ClassChain contains the self-class and its hierarchically related classes in root-to-leaf order.

The ClassChain is sub-divided into the following three components:

1. **Ancestor-Component** - contains the classes that are ancestors of the self-class. Contained within this component is the **Parent-Component** which contains the classes that are the self-class’ parents.

2. **Class-Component** - contains the self-class only.

3. **Descendant-Component** - contains the classes that are descendants of the self-class. Contained within this component is the **Child-Component** which contains the classes that are children of the self-class.
Figure 5.3: ClassChain Components and Usage

Figure 5.3a shows the ClassChain components associated with self-class C4. Figure 5.3b illustrates how these components are exploited to evaluate the hierarchical XPath axes:

- A combination of the Class-Component and the Ancestor-Component is used for the ancestor and ancestor-or-self XPath axes.

- Components Parent-Component and Class-Component are used for the parent axis.

- Components Class-Component and Child-Component are used for the child axis.

- Components Class-Component and Descendant-Component are used for the descendant axis.

The Parent-Component and the Child-Component are optional indexing constructs. They provide additional performance across the parent and child axes because they eliminate a larger number of branch classes from the query process.
5.2.1 Modelling the Indexing Constructs

In this section, we describe how the optimisation constructs are modelled. The Class Diagram in Figure 5.4 shows that each Node has a NodeLabel. A NodeLabel captures a node’s relationship to other nodes. In this instance, we chose preorder and postorder labels (i.e. pre/post encoding). However, as our approach is encoding scheme independent, all XML encoding schemes are permitted in the NodeLabel. For example, ORDPATH encoding [55] is more appropriate for systems that require frequent updates, whereas pre/post encoding is best suited to read only XML databases. A Node has the additional attributes: name (the node’s name), value (text values), type (element/attribute), and level (distance from the document node), which capture its properties within the document.

![Class Diagram](image)

Figure 5.4: Optimisation Constructs

In the storage model, one or more nodes belong to a single branch instance and a branch instance has one or more nodes (its nodeList[]). Each branch instance has a classify() function that places the branch in the relevant branch class. Thus, id in Branch will be the branch class identifier of the branch after it has been classified. In the ClassChain (Figure 5.3), this id is
the Class-Component. In the Branch class the two attributes ancestorList[] and descendantList[] contain the set of ancestor branch instances and descendant branch instances respectively (before the classify() function has been called). After classification, they contain the ancestor branch classes (Ancestor-Component in the ClassChain) and descendant branch classes (Descendant-Component in the ClassChain) respectively.

5.2.2 Worked Example

We will now provide a worked example to illustrate how optimisation is achieved for hierarchical XPath steps using the BranchClassIndex. Figure 5.5 depicts a small segment taken from the bicycle rental repository. Each node in the XML tree is identified by its preorder label. The branch class for each node is also shown; for example, nodes 2, 3, 4, 15, 16, and 17 are in branch class C5.

![Figure 5.5: Bicycle Rental Repository Subset](image)

The following example contrasts our query process to that of similar approaches [29, 31]. The goal is to illustrate the two key concepts of the BranchClassIndex: (1) search space pruning and (2) node bypassing.

**Example 9** Return all bicycle stations in Dublin.

//Dublin/stations/station
The XPath query in Example 9 contains three steps, each of which locates nodes within the XML tree as follows:

//Dublin. The first step contains the descendant-or-self axis. As it is the first step in the expression, the nodes that are descendant (or self) of the document node (node 0) must be located. The NodeTest specifies that those nodes must be element nodes that have the name Dublin.

- **Traditional Approach.** Identify a sequence of element nodes called Dublin, which is the sequence of nodes: (2, 15) - see Table 5.3.

- **BranchClassIndex Approach.** The proxy for class C5 in the NCLT index is shown in Table 5.4. Thus, in an instance of node bypassing, the NCLT index is exploited to identify a single proxy: p1, which represents both node 2 and node 15. Proxies p1-p3 are placed in Table 5.4 for illustration purposes only; these identifiers do not appear in the index.

Essentially, in the first XPath step, the traditional approach identifies two nodes in the node repository, whereas the BranchClassIndex approach identifies a single proxy node in the much smaller NCLT index that represents both nodes.

/stations. Identify the element nodes called stations that are children of the nodes identified at the first step.

<table>
<thead>
<tr>
<th>pre</th>
<th>post</th>
<th>name</th>
<th>class</th>
<th>level</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>Dublin</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>stations</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>station</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>19</td>
<td>Dublin</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>stations</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>17</td>
<td>station</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.3: Extract from the Node Repository (Base Data)
• **Traditional Approach.** The context node sequence is (2, 15). Therefore, using *pre/post* labels (see Table 5.3), node 2 is evaluated to find all nodes in the node repository that have a preorder identifier greater than that of node 2 and a postorder label less than that of node 2. Then, make sure it is a child node using the node’s *level* attribute - the node is a child if it resides at one level greater than that of the context node. The process is then repeated for node 15. The target node sequence (4, 17) is identified.

• **BranchClassIndex Approach.** This is the first step in which the ClassChain is utilised. As the step contains the child axis, ClassChain components: 2 (Class-Component) and 3(a) (Child-Component) are selected (recall ClassChain component usage Figure 5.3); these components contain branch classes 4 and 5 (C4 and C5 in Figure 5.5). It must now be determined if it is necessary to access the base data or to use the NCLT covering index. The query process uses these ClassChain components to *prune the search space* in either case because all classes that are not in the class chain are avoided. The index selection process is detailed in Chapter 6; in this particular instance, it will choose the NCLT index.

The input to this step was the sequence of proxy identifiers: (p1). Thus, using the attributes of proxy p1 (Table 5.4), in an instance of *node bypassing*, the query process will query the NCLT index and find the nodes called (*Name*) stations; that are in (*Class*) class 4 or 5; that have a level (*Level*) greater than that of Dublin (i.e. must be greater

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>class</th>
<th>level</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>Dublin</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>p2</td>
<td>stations</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>p3</td>
<td>day</td>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.4: Extract from the NCLT Covering Index
than 2); and finally \((Type)\), it must be of type 1 (an element node).
The output is therefore the sequence \((p2)\), which represents both node 4 and node 17.

The traditional approach processes each node in the context node sequence and identifies all target nodes for every step in the expression. In contrast, for the BranchClassIndex, the query process receives a sequence of proxy representatives and (1) prunes the search space using the ClassChain, and (2) bypasses nodes by identifying a sequence of target proxy representatives, rather than a sequence of target nodes. In this instance, the sequence \((p2)\) represents nodes 4 and 17 in the base data. In practice, a proxy will often (see Chapter 7 for details) represent thousands and even millions of nodes in large XML datasets.

/station. Identify the element nodes called: station that are children of the nodes identified at the previous step.

- **Traditional Approach.** The context node sequence is \((4, 7)\). Again, each node is evaluated in turn and using the context node’s attributes (e.g. preorder, postorder, level); target nodes \((5, 18)\) are identified. As this is the rightmost step in the expression, nodes 5 and 18 are returned.

- **BranchClassIndex Approach.** The sequence of context proxies contains \(p2\). As it is the rightmost step in the expression, the NCLT index cannot be used as the actual nodes must be retrieved and returned as the result set. In other words, node bypassing is not possible for the rightmost step in any XPath expression (see index selection - Chapter 6 - for more details).

However, the ClassChain can be exploited to prune the search space. As the axis is \(child\), ClassChain components associated with the classes
in the context node sequence (p2) must be identified. Thus, the components that are associated with class 5 (C5 - Figure 5.5) are selected: 2 (Class-Component) and 3(a) (Child-Component), which contain classes 4 and 5 (C4 and C5 Figure 5.5). It remains to find all nodes in the base data that have the name station; are in class 4 or 5; have a level one greater than 3; and a type of 1 (an element node). The node set \{5, 18\} is therefore returned.

5.2.3 Worked Example Summary

The result set for the XPath expression in Example 9 is \{5, 18\} and both approaches returned the same result. However, the traditional approach visited nodes (2, 4, 5, 15, 17, 18) using inefficient pre/post/level node comparisons. In contrast, the BranchClassIndex approach evaluated proxies (p1, p2) - which represented nodes (2, 4, 15, 17) - and the search space was pruned using the class chain in each case. Finally, nodes 5 and 18 are the result nodes for the query, thus the NCLT index could not be used. When locating these nodes in the base node repository, the BranchClassIndex pruned the search space by evaluating only those nodes in classes C4 and C5 - all other branch classes were avoided.

5.3 Extending the Branch Classification Process

In this section, we show how some XPath steps can be optimised by extending the classification process to include text values (recall that they were omitted from the classification process to minimise the number of branch classes). However, we only allow certain text values to be classified (those that do not increase the size of the index significantly). Using the same terminology as in [47], these text values can be categorised as high selectivity and low selectivity text values.
If a text value has high selectivity, it will occur a small number of times in the base data. Thus, in spite of the fact that the base data must be evaluated (when an XPath query is predicated on a text value) text values that have high selectivity lead to fewer node comparisons then if they have low selectivity. In contrast, when an XPath expression is predicated on a text value that has low selectivity, there will be a large number of inefficient node comparisons. This is because text values that have low selectivity occur many times in the base data.

Figure 5.6: XML Tree Showing Branch Classifications

To optimise the BranchClassIndex for queries that evaluate text nodes that have low selectivity, the paths to certain text values are included in the classification process. For example, in Figure 5.6, notice the how text values iValue1 and jValue1 are within SubDocument 1, which means that the paths to these text values are included when calculating the forward paths of element nodes in that sub-document. This has the effect of placing branches B1 and B2 in different branch classes, whereas before (see Figure 5.1) they were placed in the same branch class.

As we demonstrate empirically in Chapter 7, many text values that have low selectivity can be classified while introducing only a small number of additional branch classes. This has the effect of optimising XPath expres-
sions text values that have low selectivity for only a small increase to the index size. Our goal is to create a situation where the only time the base data is evaluated will be to evaluate text values that have high selectivity. Therefore, improving the generic performance of the index and providing a lever to control the space/time (index size/query performance) trade off.

To illustrate why optimisation is only achieved through the classification of text values that have low selectivity, consider the two queries in Figure 5.7. Query Q3 must access the base data at step two because the text value ‘17’ must be evaluated. In a real world scenario, there can be many thousands of people living in the same postcode (e.g. at postcode 17). Thus, there can be thousands of node comparisons as each person residing at postcode 17 will be evaluated. In contrast, it is unlikely (see Q4) that there will be thousands of people with identical names, i.e. M. Kersten, living in the same postcode. In this situation, even though the base data must be accessed in step two of Q4, there may be few node comparisons required (just one for each person with the name M. Kersten).

### 5.3.1 Identifying Text Values that have Low Selectivity

In order to classify text values that have low selectivity, the classification process must be able to detect them. We propose two ways in which information relating to these text values can be retrieved in advance of the classification process:

---

**Figure 5.7: Illustrating the Effect of Low Cardinality Text Values**

To illustrate why optimisation is only achieved through the classification of text values that have low selectivity, consider the two queries in Figure 5.7. Query Q3 must access the base data at step two because the text value ‘17’ must be evaluated. In a real world scenario, there can be many thousands of people living in the same postcode (e.g. at postcode 17). Thus, there can be thousands of node comparisons as each person residing at postcode 17 will be evaluated. In contrast, it is unlikely (see Q4) that there will be thousands of people with identical names, i.e. M. Kersten, living in the same postcode. In this situation, even though the base data must be accessed in step two of Q4, there may be few node comparisons required (just one for each person with the name M. Kersten).
1. **Domain Knowledge.** Knowledge of the data is known in advance.

For example, if one knows that the dataset contains an element called `gender`, the classification process can be told to include text values that have low selectivity such as `male` and `female`.

2. **Text Value Identification Algorithms.** These algorithms will identify text values that are suitable for classification.

The first approach requires no further details, because prior knowledge of data will already be available by some other means. Therefore, we will focus on the second (identification algorithms) approach, which is applicable to all XML datasets.

**Q5:** `//people//person[.//address//postcode = '17']//profile/education`

**Q6:** `//people//person[.//address//postcode > '17']//profile/education`

Figure 5.8: Examining Text Value Operations

There are a couple of considerations to be aware of when classifying text values that have low selectivity. For example query Q5 in Figure 5.8, specifies that all postcodes with a text value *equal to* 17 are evaluated. Conversely, in Q6 the *range* operator (‘\(>')\') requires that all postcodes *greater than* 17 be evaluated. In order to perform range evaluations across text values associated with a given node, *all* of the text values for that node must be classified (not just individual text values). In other words, it is not sufficient to classify the text value ‘17’ for the element node `postcode`; all text values associated with `postcode` must be classified.

Classifying all of the text values associated with a given node can lead to a large increase in the number of branch classes and therefore, the size of the overall index. In contrast, for evaluations based on the equality operator (‘\(=\)’), it is sufficient to classify individual text values associated with a given node (all of its text values do not have to be classified). Thus, the process
for identifying text values that have low selectivity is broken into two parts:

1. **Range Based Classification.** The classification of all text values associated with a node if there is a small number of distinct text values for a large number of nodes. For example, there can be a large number of nodes called *gender* with only two distinct values *male* and *female*. In this instance, all of the values associated with nodes called *gender* are classified.

2. **Equality Based Classification.** Classification of individual text values that have a cardinality greater than some threshold (thresholds are discussed below). Only the equality operator (’=’) will be permitted in XPath steps that evaluate these text values (range based text value evaluations must still access the base node repository).

### 5.3.2 The Text Value Identification Algorithms

Firstly, the algorithm for identifying text values that are candidates for range based classification is provided (IdentifyRangeBasedCandidates). Secondly, the IdentifyEqualityBasedCandidates algorithm shows how individual text values that are suitable for classification are identified. A single pass of the XML document is required in advance of the classification process to identify the input values for the algorithms. The performance of both algorithms and the pass of the XML document are discussed at the end of this section.

**The Range Based Classification Algorithm**

As discussed, to support range based evaluations across text values, all of the text values associated with a node of a given name and type (element/attribute) must be classified. The IdentifyRangeBasedCandidates process is designed to identify nodes that can be classified while incurring
only a small increase to the number of branch classes.

Algorithm 2 IdentifyRangeBasedCandidates

1: NodeMap := (NodeName ↦→ TextValues);
2: NodeCardinalityMap := (NodeName ↦→ Cardinality);
3: for NodeName in NodeMap do
4:   CardinalityOfTextValues := NodeMap.get(NodeName).size();
5:   CardinalityOfTotalNodes := NodeCardinalityMap.get(NodeName);
6:   if CardinalityOfTotalNodes ≥ (CardinalityOfTextValues * N) then
7:     classify all text values associated with NodeName;
8:   end if
9: end for

As stated earlier, the text value identification processes require one additional pass of the XML document in advance of the classification process. In IdentifyRangeBasedCandidates, lines 1 and 2 are populated during this pass of the XML document. NodeMap (line 1) is a map data structure containing each NodeName mapped to its set of TextValues. For example, if the XML document contains a node called gender, after document pass the NodeMap could contain the entry: gender ↦→ {male, female}. Similarly, the NodeCardinalityMap contains each NodeName mapped to its Cardinality. Thus, if there were 23 males and 48 females in the XML document, the NodeCardinalityMap would contain: gender ↦→ 71.

For range based classification, we are only interested in nodes that have a small number of distinct text values (such as male, female) for a large number of nodes (such as gender) because their text values will have low selectivity. Thus, to optimise the document pass, we quickly eliminate nodes that have too many distinct values. For example, if an upper threshold of 100 distinct values is used, at the end of the document pass, the NodeMap and the NodeCardinalityMap will only contain nodes that have fewer than (or equal to) 100 distinct values. We discuss suitable thresholds in Chapter 7.

To determine if there is a small number of distinct text values for a large
number of nodes, for each node of a given name (NodeName) in the XML document (line 3), the process identifies its total number of distinct text values (CardinalityOfTextValues) (line 4) and the total number of nodes with that name (CardinalityTotalNodes - line 5).

If the total number of nodes (CardinalityOfTotalNodes) that have a given name and type is small, there is no point in classifying its text values. To see why, consider the node gender again. If there are only 100 nodes called gender in the document, there is no point in classifying the text values associated with gender, even if there are only two values (male and female).

The reason for this (as discussed earlier) is that even if the base data must be accessed to evaluate nodes called gender, it will be efficient because there will be a maximum of 100 node comparisons.

To determine if the selectivity of the text values are low enough, a threshold $T$ must be known. $T$ is time it takes to perform a join between nodes in the base data, when an XPath query is predicated on a text value. Say $CN$ (Context Nodes) $\times TN$ (Target Nodes) is the total number of node joins that can be performed when $T$ is equal to 1000ms (one second). If we classified all text values that occur more than $N$ times in the base data, all of the text values that are not indexed will occur less than $N$ times in the base data. Therefore, we would expect that any time the base data must be evaluated (because of a text value in an XPath step) the join operation would be performed in less than one second (we identify suitable $T$ values empirically in Chapter 7).

For example, if there are 10,000 nodes called gender which have two distinct values: male and female, line 6 in IdentifyEqualityBasedCandidates would be as follows: if $10,000 \geq (2 \times T)$. If for example, $T$ is set to 3000, at line 7 the text values associated with nodes called gender will be classified. In contrast, if CardinalityOfTotalNodes was 5,000, text values associated with gender would not be classified.
The Equality Based Classification Algorithm

A node can have associated text values that have low selectivity while also
having text values that have high selectivity. In this instance, range based
classification is not suitable because classifying all of the node’s text values
(including those with high selectivity) could lead to a large increase to the
index size. However, it is easy to determine if the XPath step is performing
an equality evaluation or a range based evaluation across a text value. Thus,
we can extend the index further (and avoid the base data for a greater num-
ber of XPath steps) by classifying text values that are suitable for equality
evaluations only.

Algorithm 3 IdentifyEqualityBasedCandidates

During the pass of the XML document, a NodeMap is populated which con-
tains each NodeName mapped to another map that contains its text values
mapped their cardinality (line 1). For each node name in the NodeMap
(line 2) and for each of its text values (line 3) the cardinality of the text
values (TextValueCardinality) is identified (line 4). If TextValueCardinality
is greater than some threshold \( N \) (line 5), it is classified for equality eval-
uations in XPath expressions (line 6). In other words, if the number of times
the text value occurs in the XML document is greater than \( N \) (again this \( N \)
value is determined empirically in Chapter 7), it has low selectivity.
Efficiency Evaluation

We will now discuss the efficiency of the \texttt{IdentifyRangeBasedCandidates} and \texttt{IdentifyEqalityBasedCandidates} processes.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Platform</th>
<th>CPU</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dell</td>
<td>Windows 7 Pro (32bit)</td>
<td>3.00GHz Intel Core Duo</td>
<td>4GB</td>
</tr>
</tbody>
</table>

Table 5.5: System Specification

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Pass Time</th>
<th>IRBC</th>
<th>IEBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bicycle Rental</td>
<td>2.06 GB</td>
<td>2.7 min</td>
<td>1 ms</td>
<td>11 ms</td>
</tr>
<tr>
<td>DBLP</td>
<td>676 MB</td>
<td>0.7 min</td>
<td>0 ms</td>
<td>18 ms</td>
</tr>
<tr>
<td>XMark</td>
<td>1.33 GB</td>
<td>0.8 min</td>
<td>1 ms</td>
<td>20 ms</td>
</tr>
<tr>
<td>Protein</td>
<td>683 MB</td>
<td>0.6 min</td>
<td>0 ms</td>
<td>28 ms</td>
</tr>
</tbody>
</table>

Table 5.6: Efficiency of the Text Value Identification Algorithms

The processes were executed on a system with the specification shown in Table 5.5 and they were implemented in the Java programming language. Table 5.6 shows four datasets, their size, the document pass time in minutes, the \texttt{IdentifyRangeBasedCandidates} (IRBC) time in milliseconds, and the \texttt{IdentifyEqalityBasedCandidates} (IEBC) time in milliseconds respectively. The largest quantity of time is spent on the document pass, which identifies the input values for both processes. After the document pass is complete, the values in Table 5.6 show that the processes take very little time (less than one second for each dataset).

5.4 Post Classification Integrity

At this point, we have described the branch classification process in full. Additionally, in §5.2, we described how branch classes are used in the query process, that is, how the ClassChain components are utilised at each step in an XPath expression. The objective of this section is to ensure that the integrity of query result sets are maintained after branch classification.
The BranchClassIndex works on the basis that \textit{name}, \{\textit{branch classes}\}, \textit{level}, \textit{type}, \textit{value} (i.e. the NCLTV index) can be used to evaluate hierarchical XPath steps. The NCLTV index contains a \textit{branch class proxy} that represents many branch instances. Thus, as the NCLTV index is used in place of individual node labels such as \textit{pre/post}, we must ensure that the following IntegrityProposition holds:

\textbf{IntegrityProposition}. If a single branch instance in a branch class contains target node(s), all branch instances in that class will contain equivalent target node(s).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{xml_tree.pdf}
\caption{XML Tree Showing Branch Classifications}
\end{figure}

The use of the word \textit{equivalent} in referring to one or more nodes in the following deductions means: a \textit{set} (sorted in document order) of nodes that have the same \textit{name}, \textit{type}, \textit{level} and \textit{value} (\textit{value} only includes the text values that were selected for range or equality based classification).

\textbf{Example 10 (Sample Linear XPath Expression)}

\texttt{/a//c//h/ancestor::f.}
Case 1 (XPath Expressions). Given any linear XPath expression containing one of the six hierarchical XPath axes (such as Example 10), we can deduce that if a single branch instance (B3 Figure 5.9) contains a target node (node 21), then all branch instances in the same class (B4) will also contain a target node (nodes 30). The reason for this is that based on the branch classification, all branch instances have equivalent forward and backward structure.

Example 11 (Sample XPath Twig Query with Text Value)
/a//i[./@key = 'iValue1']/ancestor::f.

Case 2 (Twig Query with Text Value). When a Twig query contains a low selectivity text value that has been selected for classification, the path containing nodes from the root node to the text value itself is included in the classification process. For example, branches B1 and B2 are placed in different branches because text values ’iValue1’ and ’jValue1’ were classified. Thus, for Example 11, the result node is node 5; node 13 was not returned because B1 and B2 have different forward and backward structure.

5.5 Branch Classification Summary

In this chapter, we described how the BranchIndex can be compacted. To achieve this, we showed how branch instances can be classified with the result that the performance benefits of the BranchIndex can be achieved for a fraction of the storage (and therefore) index traversal costs. This specialised version of the BranchIndex is called the BranchClassIndex. The classification process was then detailed and a description of the query process that exploits the BranchClassIndex for query performance was provided. For this purpose, the concept of a ClassChain was introduced. In particular, we illustrated how the ClassChain was used to provide a search space pruning
facility. We then introduced node bypassing based on the NCLT covering index. Finally, it was shown how the classification process can be extended to include text values that have low selectivity (to produce the NCLTV index). This leads to overall performance gains because the number of node comparisons that will be required when the base data must be accessed is reduced.
Chapter 6

Query Processing

At this point, we have presented a node partitioning process for XML documents and defined those partitions as *branches*. A description of how the search space can be pruned by indexing branch instances was also provided. In practice however, we showed how may be too many branch instances (in an XML document) to achieve significant optimisation. Therefore, a classification process for branch instances was provided in chapter 5. We demonstrated how the resulting BranchClassIndex, which is a specialised version of the BranchIndex, achieves search space pruning for reduced storage costs. Additionally, it was demonstrated that the properties of branch instances (that are implicit within the indexed branch classes) can be exploited to facilitate the concept of *node bypassing* using the NCLTV covering index for XPath steps.

This chapter now describes how these concepts can be exploited to optimise XPath expressions (processes p11, p13 and p14). There are two ways of deploying this storage model, native XML indexing (non-relational) or XML-enabled (relational) indexing. The second was chosen because, as we will show, the BranchClassIndex is suitable for deployment in an off-the-shelf relational database, the many advantages of which were discussed in Chapter 2. Relational index deployment is detailed in §6.1. In §6.2, the
XPath-to-SQL transformation is presented. This is followed in §6.3, with a worked example to illustrate the transformation process. Finally, an index selection process is required to determine when the base data must be evaluated and when the covering index can be exploited; this index selection process is presented in §6.4.

6.1 Index Deployment

Figure 6.2 illustrates how the XML document in Figure 6.1 is transformed into four relations. The Node relation in Figure 6.2b1 is the base data (construct 16 in the system architecture). Attributes `pre` and `post` are the preorder and postorder labels of each node respectively. Attribute `name` is the node’s name; `type` differentiates between element attribute and document nodes; `level` is the node’s distance from the document node; and `value` is the text value associated with the node (or `null`).

The NCLTV relation (6.2b2) is populated by selecting distinct `name`, `class`, `level`, `type` and `value` from the Node relation (but `value` is included only for low selectivity text values). The `class` attribute in the Node and NCLTV relations is a node’s `branch class identifier`.

The Parent and Child ClassChain components are deployed in the PC_REL relation (6.2b3). The AD_REL relation (6.2b4) contains the Ancestor and Descendant components. In the PC_REL relation, attribute `pc` contains the parent-and-self branch classes (Parent-Component and Class-Component combination) and attribute `cc` contains the child-and-self branch classes (Child-Component and Class-Component combination). In the AD_REL relation, the attribute `ac` is the ancestor-and-self branch classes (Ancestor-Component and Class-Component combination) and `dc` is the descendant-and-self branch classes (Descendant-Component and Class-Component combination).
Figure 6.1: XML Snippet from the Bicycle Rental Dataset
Figure 6.2: Node Relation and the NCLTV Index Relations
6.2 The Transformation Method

In this section, we describe how an XPath expression is transformed into its SQL equivalent. We begin with a description of our Transformation Template and then describe the method used to construct the SQL expression using this template to extract the necessary information from the XPath query expression.

The indexing constructs are used to improve the performance of XPath steps that contain one of the six hierarchical axes. Although optimisation specific to other axes is not currently provided, we discuss the future inclusion of these axes (such as following and preceding) as part of our future work. Thus, the axes shown in Table 6.1 constitute the subset (the most commonly used axes) of the XPath 2.0 language covered here.

<table>
<thead>
<tr>
<th>XPath Axis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ancestor</td>
<td>Ancestors of ( x ).</td>
</tr>
<tr>
<td>ancestor-or-self</td>
<td>Node ( x ) and its ancestors.</td>
</tr>
<tr>
<td>descendant</td>
<td>Descendant of ( x ).</td>
</tr>
<tr>
<td>descendant-or-self</td>
<td>Node ( x ) and its descendents.</td>
</tr>
<tr>
<td>parent</td>
<td>The parent of ( x ).</td>
</tr>
<tr>
<td>child</td>
<td>The children of ( x ).</td>
</tr>
</tbody>
</table>

Table 6.1: XPath 2.0 Language Coverage

The transformation template (Figure 6.3) represents an SQL expression, which we refer to as the BranchClass expression. It is divided into sub-expressions. These sub-expressions are populated from an expression using a series of transformations rules, which are now described.

<table>
<thead>
<tr>
<th>Name</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>SELECT</td>
</tr>
<tr>
<td>C2</td>
<td>( c_{n-1}, \text{classop} )</td>
</tr>
<tr>
<td>C3</td>
<td>FROM</td>
</tr>
<tr>
<td>C4</td>
<td>([\text{indexop } i_x, \text{CLASS } c_x] +)</td>
</tr>
</tbody>
</table>

Table 6.2: The Generic Expression Components
Variables

classop ::= ac|dc -- class index column options
equalop ::= >|=|<|≤|≥ -- equality options
indexop ::= NODE|NCLT -- index options

1. SELECT * FROM NODE step,
2. WHERE [step.NAME = 'name-prop' AND] step.TYPE = 'type-prop'
3. [AND step.LEVEL = 0]?
4. AND step.CLASS IN ( SELECT cn−1.classop FROM [indexop ix, CLASS cx]+
5. WHERE [ix.NAME = 'name-prop' AND? ix.TYPE = 'type-prop'
6. [AND ix.LEVEL = 0]?
7. AND c1.classop = i1.CLASS
8. [AND EXISTS ( <predicate-statement> ) ]*
9. [ AND [i1.NAME = 'name-prop' AND? i1.TYPE = 'type-prop'
10. AND i1.CLASS = c1.classop
11. [AND i1.LEVEL equalop i1−1.LEVEL [adlevel]]? ]
12. [ i1.PRE equalop i1−1.PRE ∧ i1.POST equalop i1−1.POST ]?
13. [AND EXISTS ( <predicate-statement> ) ]*
14. [ AND EXISTS ( <predicate-statement> ) ]*
15. [ AND EXISTS ( <predicate-statement> ) ]*
16. [ AND EXISTS ( <predicate-statement> ) ]*
17. ORDER BY PRE

<predicate-statement> ::= 21 SELECT null FROM indexop pstep,
22 line 2
23 line 3
24 line 4
25 SELECT p(n−1).classop FROM CLASS p1, [indexop pi, CLASS px]*
26 WHERE p1.classop = indexIn.CLASS
27 [lines 10-15]*
28 line 16
29 line 17
30 line 18
31 line 19

Figure 6.3: The XPath-to-SQL Transformation Template
6.2.1 Generic Expression

Within the transformation template and unlike all other expressions, the generic expression is constructed using properties from all steps in the XPath expression. Its four components are shown in Table 6.2. We will now describe how the generic expression is populated.

In the SQL expression, the required tuples are selected from relevant relations using the keywords in components C1 and C3. The tuples that must be returned are specified by C2 and the relations that contain these tuples are specified in C4. The classop (class options variable) in C2 is populated by looking ahead to the last step in the XPath expression and extracting its axis. The LastAxisProperty (the axis at the rightmost step) is used for this purpose:

**LastAxisProperty.** The XPath axis associated with the last step. This enables the branch classes that must be evaluated at the last step to be identified. Using the LastAxisProperty, the class options are as shown in Table 6.3.

<table>
<thead>
<tr>
<th>classop</th>
<th>Selected when...</th>
</tr>
</thead>
<tbody>
<tr>
<td>dc</td>
<td>LastAxisProperty is the: descendant, descendant-or-self, or child axis.</td>
</tr>
<tr>
<td>ac</td>
<td>LastAxisProperty is the: ancestor, ancestor-or-self, or parent axis.</td>
</tr>
</tbody>
</table>

Table 6.3: Populating the Class Option Variable

As shown in Table 6.3, if the LastAxisProperty is the descendant axis, the classop will be dc, thus C2 will be: \( c_{n-1}.dc \). Therefore, the Descendant-Component and Class-Component combination (i.e. the descendant-or-self branch classes of the previous step) will be the only branch classes evaluated at the last step.

For the first step in the XPath expression, the Node or the NCLTV relation...
must be accessed to identify the first set of context nodes or their proxies, respectively. The index selection process that chooses between these two relations is detailed in §6.4. For each subsequent step (excluding the rightmost), the ClassChain components are used to prune the search space. Then, the \texttt{Node} or the \texttt{NCLTV} relation is evaluated to locate the target nodes (or their proxies). Finally, the ClassChain components are used one more time (to return the relevant branch classes in \( C_2 \)). In other words, component \( C_4 \) is repeated \( n-1 \) times, i.e. one less than the number of steps in the expression\(^1\).

Component \( C_4 \) contains two variables. The first variable: \textit{indexop} (index options) is populated with \texttt{Node} or \texttt{NCLTV}. The second variable: \textit{x} denotes the number of repetitions of \( C_4 \). For example, if there are 5 steps in the expression, \( C_4 \) is repeated 4 times. Thus, the \textit{x} variable will be 1 for the first repetition of \( C_4 \), 2 for the second and so on (up to 4).

\subsection*{6.2.2 Transforming the Initial XPath Step}

This Initial Step expression (see Template - Figure 6.3) maps directly to the first step in the XPath expression. It filters those tuples associated with aliases \( i_1 \) and \( c_1 \) (these aliases were discussed for \( C_4 \) above). In order to transform the first step in an XPath expression, three additional properties (AxisProperty, NameProperty and TypeProperty) are extracted from the first step in the XPath expression. We will now describe the method that uses these properties to construct the Initial Step expression.

\textbf{AxisProperty.} The XPath axis. The supported axes are: \textit{ancestor}, \textit{ancestor-or-self}, \textit{parent}, \textit{descendant}, \textit{descendant-or-self} and \textit{child} (the hierarchical XPath axes).

\footnote{Square brackets and other BNF symbols (i.e. ‘+’ and ‘?’) are not part of the SQL statement.}
**NameProperty.** The *name* of the target nodes. The name property can also be an XPath wildcard (denoted by `*`).

**TypeProperty.** The *type* of the target node. The type of the target node can be element or attribute.

**NextAxisProperty.** The XPath axis associated with the next step in left-to-right order. This enables us to identify the branch classes that must be evaluated at the next step. This is crucial in managing the dependency between steps in an XPath expression. Using the `NextAxisProperty`, the class options are as shown in Table 6.5.

<table>
<thead>
<tr>
<th>Name</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>C5</td>
<td>WHERE</td>
</tr>
<tr>
<td>C6</td>
<td>[i1.NAME = ‘name-prop’ AND]?</td>
</tr>
<tr>
<td>C7</td>
<td>i1.TYPE = ‘type-prop’</td>
</tr>
<tr>
<td>C8</td>
<td>[AND i1.LEVEL = 0]?</td>
</tr>
<tr>
<td>C9</td>
<td>AND c1.classop = i1.CLASS</td>
</tr>
<tr>
<td>C10</td>
<td>[AND EXISTS (&lt;predicate-statement&gt;)]*</td>
</tr>
</tbody>
</table>

Table 6.4: The Initial Step Expression Components

The Initial Step expression (lines 6-9 in the transformation template) contains the components shown in Table 6.4. The first component (C5) is the *where* clause, which is required. The where clause filters the *sets* of nodes represented by aliases ‘i1’ and ‘c1’ (in component C4). Component C6 may be included zero or one times and has one variable (name-prop). If the NameProperty is a *wildcard* C6 is omitted (so that nodes of any name are evaluated), otherwise the name-prop variable is populated with NameProperty. Component C7 is compulsory and contains a single variable type-prop, which is populated with the TypeProperty.

An XPath expression that begins with ‘/’ evaluates the document node at the first step. The document node is the only node that resides at *level*
Selected when...

<table>
<thead>
<tr>
<th>classop</th>
<th>NextAxisProperty is the:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac</td>
<td>descendant, descendant-or-self, or child axis.</td>
</tr>
<tr>
<td>dc</td>
<td>ancestor, ancestor-or-self, or parent axis.</td>
</tr>
</tbody>
</table>

Table 6.5: Populating the Class Option Variable

0 in our system. Thus, in this situation component C8 is included to ensure that the document node is the only node that is evaluated. If the XPath expression begins with ‘//’, nodes at all levels must be evaluated, i.e. descendant-or-self nodes of the document node, thus C8 is omitted.

Component C9 is used to set the branch classes that are related to the set of nodes located by step one in the XPath expression because their hierarchically-related branch classes will be required at the next step. Component C9 contains a single variable classop, which is populated based on the NextAxisProperty, the class options in Table 6.5.

In XPath, a predicate filter removes some of the nodes identified at its associated step and leaves others [68]. An XPath step may have one or more predicate filters. Component C10 is used for each such filter. As component C10 is generic to all steps (it appears at lines 9, 15 and 19 respectively in the Transformation Template) we will detail C10 later in §6.2.5.

In instances where the XPath expression contains only a single step, the process described in this section and the processes described in §6.2.1 and §6.2.3 do not take place. Instead, processing moves directly to the final step transformation described in §6.2.4.

6.2.3 Transforming Interim Steps

The Interim Step expression is used in the transformation of the second step through to the second last step. With the exception of variable x (the step number) the expression at line 10 is identical to components C5–C7. Line 14 is identical to C9 and line 15 is identical to C10. Therefore, it is
not necessary to repeat the explanations. The \( x \) variable is incremented for each step. For example, in the second step \( x \) will be 2. The components within the interim step expression (lines 11, 12 and 13 in the template) are shown in Table 6.6.

<table>
<thead>
<tr>
<th>Name</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>C11</td>
<td>( \text{AND } i_x.CA_{\text{SS}} = c_{x-1}.\text{classop} )</td>
</tr>
<tr>
<td>C12</td>
<td>([\text{AND } i_x.LEVEL \text{ equalop } i_{x-1}.\text{LEVEL } [\text{adlevel}]?])</td>
</tr>
<tr>
<td>C13</td>
<td>([i_x.PRE \text{ equalop } i_{\text{lnaStep}}.\text{PRE } &amp; \ i_x.POST \text{ equalop } i_{\text{lnaStep}}.\text{POST}])</td>
</tr>
</tbody>
</table>

Table 6.6: The Interim Step Expression Components

<table>
<thead>
<tr>
<th>classop</th>
<th>Selected when...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( dc )</td>
<td>( \text{AxisProperty is the:} )</td>
</tr>
<tr>
<td></td>
<td>( \text{descendant}, \text{descendant-or-self}, \text{or child axis.} )</td>
</tr>
<tr>
<td>( ac )</td>
<td>( \text{AxisProperty is the:} )</td>
</tr>
<tr>
<td></td>
<td>( \text{ancestor}, \text{ancestor-or-self}, \text{or parent axis.} )</td>
</tr>
</tbody>
</table>

Table 6.7: New Class Options

Component C11 locates the ClassChain components that must be examined. In C11, variable \( x \) is the step number and \( x-1 \) denotes the previous step number. In this instance, the \( \text{classop} \) variable is populated based on the \( \text{AxisProperty} \) as shown in Table 6.7. Component C12 is required if the \( \text{AxisProperty} \) is the parent or child axis (to differentiate between ancestor/parent and descendant/child axes respectively). Additionally, C12 is required when the \( \text{indexop} \) is the \( \text{NCLTV} \) relation, in which case C13 is omitted. Based on the current XPath axis, the \( \text{equalop} \) in C12 is populated as shown in Table 6.8.

In addition to the \( \text{equalop} \) variable in C12, the variable \( \text{adlevel} \) (additional level constraint) is added for the parent and child axes. If the axis is parent the \( \text{adlevel} \) variable is populated with \(-1\); for the child axis it is populated with \(+1\). If the \( \text{indexop} \) is the \text{Node} relation, C13 must be included, but C12 omitted for all but the parent and child axes. In C13, \text{lnaStep} is the number of the last step that evaluated the \text{Node} relation (the \text{lnaStep} is detailed in
<table>
<thead>
<tr>
<th>Axis</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>ancestor</td>
<td>'&lt;'</td>
</tr>
<tr>
<td>ancestor-or-self</td>
<td>'&lt;='</td>
</tr>
<tr>
<td>descendant</td>
<td>'&gt;'</td>
</tr>
<tr>
<td>descendant-or-self</td>
<td>'=&gt;'</td>
</tr>
<tr>
<td>parent</td>
<td>'='</td>
</tr>
<tr>
<td>child</td>
<td>'='</td>
</tr>
</tbody>
</table>

Table 6.8: Equality Options for Component C12

§6.4). The two equality option (equalop) variables in C13 are shown, with ordering, based on the XPath axis in Table 6.9.

<table>
<thead>
<tr>
<th>Axis</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>ancestor</td>
<td>'&lt;' '&lt;'</td>
</tr>
<tr>
<td>ancestor-or-self</td>
<td>'&lt;=' '=&gt;'</td>
</tr>
<tr>
<td>descendant</td>
<td>'&gt;' '&lt;'</td>
</tr>
<tr>
<td>descendant-or-self</td>
<td>'=&gt;' '&lt;='</td>
</tr>
<tr>
<td>parent</td>
<td>'&lt;' '&gt;'</td>
</tr>
<tr>
<td>child</td>
<td>'&gt;' '&lt;'</td>
</tr>
</tbody>
</table>

Table 6.9: Equality Options for Component C13

6.2.4 Transforming the Final XPath Step

The Leaf Opening and Leaf Closing expressions are used in the transformation of the final (or leaf) step in the XPath query expression. This final step (step_n) is always the rightmost step in the XPath expression. In the leaf opening expression, with the exception that alias i_1 is replaced with step_n, line 2 contains components C5, C6 and C7 (explained earlier). Similarly, by substituting the same aliases (step_n for i_1), line 3 is identical to C8.

The distinct components in the leaf opening expression are C14 and C15 (Table 6.10). Component C14 is static (i.e. it contains no variables) and selects all tuples from the Node relation (the result set). If there is more than one step in the expression, the static component C15 is included to ensure that the branch classes identified at the preceding step are the only
ones that are evaluated in \( \text{step}_n \). This has the effect of providing a final pruning phase at the point of generating the final result set.

\[
\begin{array}{|c|l|}
\hline
\text{Name} & \text{Component} \\
C_{14} & \text{SELECT * FROM Node}\ \text{step}_n \\
C_{15} & \text{AND}\ \text{step}_n.\text{CLASS IN (} \\
C_{16} & \text{)} \\
C_{17} & \text{ORDER BY PRE} \\
\hline
\end{array}
\]

Table 6.10: Leaf Path Expression Components

The second requirement for transforming \( \text{step}_n \) is to construct the leaf closing expression. Within the leaf closing expression, lines 16 and 17 are identical to C12 and C13 respectively, with the exception that each occurrence of alias \( i_x \) is replaced with alias \( \text{step}_n \) and its distinct components are C16 and C17 (Table 6.10). Component C16 terminates the sub-query (which begins at C15) and C17 ensures that the result’s nodes are returned in document order, as required by the XPath [66] specification.

6.2.5 Transforming XPath Predicate Filters

Within the main transformation template, lines 9, 15 and 19 contain an EXISTS sub-expression that filters nodes based on XPath predicates. In XPath, a predicate filter contains a nested XPath expression, i.e. it consists of a number of steps and each step has an Axis, NodeTest and zero or more (sub) Predicates.

The \(<\text{predicate-statement}>\) (lines 21-31 in the transformation template) differs from the main template in two ways. Firstly, all of the alias names are given the prefix \( p \) (e.g. \( p\text{step}_n \)) so that alias names in the sub-statements do not conflict with the alias names in the main statement. Secondly, each line in the \(<\text{predicate-statement}>\) that refers back to a line in the main template is identical to that line, with the exception of the prefix on each alias. The new components within the \(<\text{predicate-statement}>\) are shown in Table 6.11.
Component C18 is required and does not contain a variable. In component C18, null is selected as we do not require the result set from the sub-expression (this is the approach used by the authors in [25]). Component C19 is also required and contains the single variable `indexop`. This forms the index selection process, described later in this Chapter, and is a crucial component of the optimisation. Component C20 is the same as C1 and C2, which were explained earlier.

Lines 6-9 do not appear in the predicate statement as these are used to identify the first set of context nodes for the XPath expression. Conversely, in an XPath predicate filter the nodes that must be filtered have already been identified; these nodes are the context nodes for the first step in the predicate filter. The ClassChain components that are associated with these context nodes are identified using component C21. The addition of C21 is the only difference between line 25 and line 5.

The ClassChain components associated with the context nodes at which the predicate is applied are identified using C22. The `classop` variable (in C22) is populated using the `AxisProperty` from the step to which the predicate filter applies and the class options from Table 6.7. The `indexIn` variable is the alias for the relation (which will be `Node` or `NCLTV` based on `indexop`) at the step to which the filter applies. For example, if the filter applies to step three in the XPath expression, `indexIn` will be: $i_3$.

<table>
<thead>
<tr>
<th>Name</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>C18</td>
<td>SELECT null</td>
</tr>
<tr>
<td>C19</td>
<td>FROM <code>indexop.pstep</code></td>
</tr>
<tr>
<td>C20</td>
<td>SELECT <code>pc_{n-1}.classop</code></td>
</tr>
<tr>
<td>C21</td>
<td>FROM CLASS <code>pc_1</code></td>
</tr>
<tr>
<td>C22</td>
<td>WHERE <code>pc_1.classop = indexIn.CLASS</code></td>
</tr>
</tbody>
</table>

Table 6.11: Components within the `<predicate-statement>`
6.3 Sample Transformation

We will now present a sample transformation to illustrate how the transformation template is populated for an XPath query (Example 12).

Example 12
/site/closed_auctions/closed_auction//keyword

6.3.1 Transforming the Generic Expression

The following property is required to populate the generic expression:

- \textbf{LastAxisProperty} is the descendant axis, therefore \textit{dc} (descendant or self branch classes) is selected.

<table>
<thead>
<tr>
<th>Name</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>SELECT</td>
</tr>
<tr>
<td>C2</td>
<td>\textit{c_3}, \textit{dc}</td>
</tr>
<tr>
<td>C3</td>
<td>FROM</td>
</tr>
<tr>
<td>C4</td>
<td>NCLT \textit{i_1}, \textit{AD_REL \ c_1}, NCLT \textit{i_2}, \textit{AD_REL \ c_2}, NCLT \textit{i_3}, \textit{AD_REL \ c_3}</td>
</tr>
</tbody>
</table>

Table 6.12: Generic Expression Components

The components in the Generic Expression are populated as shown in Table 6.12. Components \textbf{C1} and \textbf{C3} are constant (do not contain variables). The \textit{classop} variable in \textbf{C2} is populated using the \textbf{LastAxisProperty} and the options in Table 6.3. Component \textbf{C4} contains three variables: \textit{indexop} is populated based on the index selection process described later in §6.4; \textbf{CLASS} will be the \textbf{PC\_REL} (if the axis is parent or child and the optional Parent and Child ClassChain components are available) or else it will be the \textbf{AD\_REL} relation; variable \textit{x} is the step number. For our current purposes, it is assumed that the Parent and Child ClassChain components are not available, therefore \textbf{AD\_REL} is used. As there are four steps in the expression, component \textbf{C4} is repeated three times.
6.3.2 Transforming the Initial XPath Step

To transform the first step in the XPath expression, the following three properties are required:

- NameProperty = site.
- TypeProperty = 1 (element).
- NextAxisProperty is the child axis, therefore classop variable is \( \text{ac} \) (ancestor-or-self branch classes).

<table>
<thead>
<tr>
<th>Name</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>C5</td>
<td>WHERE</td>
</tr>
<tr>
<td>C6</td>
<td>( i_1 \cdot \text{NAME} = 'site' ) AND</td>
</tr>
<tr>
<td>C7</td>
<td>( i_1 \cdot \text{TYPE} = 1 )</td>
</tr>
<tr>
<td>C8</td>
<td>( \text{AND } i_1 \cdot \text{LEVEL} = 0 )</td>
</tr>
<tr>
<td>C9</td>
<td>( \text{AND } c_1 \cdot \text{ac} = i_1 \cdot \text{class} )</td>
</tr>
<tr>
<td>C10</td>
<td>( \text{AND EXISTS } &lt; \text{predicate statement} &gt; )</td>
</tr>
</tbody>
</table>

Table 6.13: Initial Step Expression Components

The components within the Initial Step expression are populated as shown in Table 6.13. The NameProperty populates the \( \text{name-prop} \) variable in C6; the TypeProperty populates the \( \text{type-prop} \) variable in C7 and the NextAxis-Property populates the \( \text{classop} \) variable in C9. There are no predicate filters at the initial step, therefore C10 is omitted.

6.3.3 Transforming Interim Steps

There are two interim steps in Example 12, which are now described.

Transforming the First Interim Step

To transform the first interim step, the following four properties are required:

- AxisProperty = is the child axis, therefore \( \text{classop} \) variable is \( \text{dc} \) (descendant-or-self branch classes).
• NameProperty = closed_auctions.

• TypeProperty = 1 (element).

• NextAxisProperty is the child axis, therefore classop variable is ac (ancestor-or-self branch classes).

<table>
<thead>
<tr>
<th>Name</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>C5</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td>C6</td>
</tr>
<tr>
<td></td>
<td>C7</td>
</tr>
<tr>
<td></td>
<td>C11</td>
</tr>
<tr>
<td></td>
<td>C12</td>
</tr>
<tr>
<td></td>
<td>C13</td>
</tr>
<tr>
<td></td>
<td>C9</td>
</tr>
<tr>
<td></td>
<td>C10</td>
</tr>
</tbody>
</table>

Table 6.14: First Interim Step Expression Components

As stated when describing the transformation process, components C5, C6 and C7 are repeated for each interim step. The only difference from their earlier description is that the step number, NameProperty and TypeProperty are updated. Component C11 is populated by selecting the branch classes associated with the AxisProperty; in C12 the $x$ variable is populated with the current step number (step 2) and the previous step number (step 1) respectively. Component C13 is omitted as there were no previous base data evaluations (see index selection §6.4 for more details). Component C9 is populated with the NextAxisProperty (as it was in the initial step transformation), and the $x$ variable is populated with step 2. There are no predicate filters at this step, thus component C10 is omitted.

**Transforming the Second Interim Step**

To populate the second interim step the following properties are used:

• AxisProperty = is the child axis, therefore classop variable is dc (descendant-or-self branch classes).
• NameProperty = closed_auction.

• TypeProperty = 1 (element).

• NextAxisProperty is the descendant axis, therefore classop variable is ac (ancestor-or-self branch classes).

Table 6.15: The Second Interim Step Expression Components

<table>
<thead>
<tr>
<th>Name</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>C5</td>
<td>AND</td>
</tr>
<tr>
<td>C6</td>
<td>$i_3$.NAME = ‘closed_auction’ AND</td>
</tr>
<tr>
<td>C7</td>
<td>$i_3$.TYPE = 1</td>
</tr>
<tr>
<td>C11</td>
<td>AND $i_3$.class = $c_2$.dc</td>
</tr>
<tr>
<td>C12</td>
<td>AND $i_3$.LEVEL = $i_2$.LEVEL+1</td>
</tr>
<tr>
<td>C13</td>
<td>$i_3$.PRE equalop $\lnaStep$.PRE ∧ $i_3$.POST equalop $\lnaStep$.POST</td>
</tr>
<tr>
<td>C9</td>
<td>AND $c_3$.ac = $i_3$.class</td>
</tr>
<tr>
<td>C10</td>
<td>AND EXISTS (&lt;predicate-statement&gt;)</td>
</tr>
</tbody>
</table>

The only difference between the first and the second interim steps is that: (1) the step numbers are updated, where $x$ is now step 3 and $x-1$ is step 2 and (2) The NameProperty is now closed_auction rather than closedauctions.

6.3.4 Transforming the Final XPath Step

To populate the final step expression, the following properties are used:

• AxisProperty = is the descendant axis, therefore classop variable is dc (descendant-or-self branch classes).

• NameProperty = keyword.

• TypeProperty = 1 (element).

In the Leaf Opening expression, components $C_5$, $C_6$ and $C_7$ are updated with the relevant step numbers and properties as shown in Table 6.16. Component $C_8$ is not required because it is not a single step XPath expression. Component $C_{14}$ completes the Leaf Opening expression.
In the Leaf Closing expression component \( C_{15} \) is constant. In component \( C_{12} \), variable \( x \) is populated with the previous step number (step 3). Component \( C_{13} \) is not required because there are no previous base data evaluations (see index selection §6.4) for more details. Finally, components \( C_{16} \) and \( C_{17} \) are static and \( C_{17} \) ensures that the result set is returned in XML document order. The completed SQL expression is shown in Figure 6.4.
-- Begin Leaf Opening Expression
SELECT * FROM Node stepN

WHERE stepN.NAME = 'keyword'
AND stepN.TYPE = 1
AND stepN.class IN ( -- End Leaf Opening Expression

-- Begin Generic Expression
SELECT c3.dc FROM NCLT i1, AD_REL c1, NCLT i2, AD_REL c2, NCLT i3, AD_REL c3 -- End Generic Expression

-- Begin Initial Step Expression
WHERE i1.NAME = 'site' AND i1.TYPE = 1
AND i1.LEVEL = 0
AND c1.ac = i1.class -- End Initial Step Expression

-- Begin First Interim Step Expression
AND i2.NAME = 'closed_auctions'
AND i2.TYPE = 1
AND i2.class = c1.dc
AND i2.LEVEL = i1.LEVEL+1
AND c2.ac = i2.class -- End First Interim Step Expression

-- Begin Second Interim Step Expression
AND i3.NAME = 'closed_auction'
AND i3.TYPE = 1
AND i3.class = c2.dc
AND i3.LEVEL = i2.LEVEL+1
AND c3.ac = i3.class -- End Second Interim Step Expression

-- Begin Leaf Closing Expression
AND stepN.LEVEL > i3.LEVEL

) ORDER BY PRE -- End Leaf Closing Expression

Figure 6.4: The Completed SQL Expression
6.4 Index Selection

Our optimization strategy includes a number of rules that determine when the index can be used, as the alternative is to access the base data to compare individual nodes. Therefore, the goal of the index selection phase is to maximize usage of the index. This is achieved through a series of rules. We now describe how the query process decides when the index can be used and when the base data must be evaluated (that is, we describe the decision making process for selecting the $\text{indexop}$ variable in the transformation).

![Diagram](image)

**Figure 6.5: Index Selection**

6.4.1 Base Index Selection Rules

When an XPath step evaluates a text value that has not been classified, the base data must be accessed. For all Twig queries that do not evaluate a text value (that has not been classified), the following index selection rules are used:

**Rule 1.** If the step being processed is the rightmost step in each XPath expression, then the base data (the $\text{Node}$ relation) must be used to evaluate the expression. This rule supersedes all other rules and remaining rules are applied only if Rule 1 does not hold.

**Rule 2.** If the step does not evaluate a text value or if the step evaluates a classified text value, then the index is used.
Rule 3. For any step that evaluates a non-classified text value, e.g. \texttt{\texttt{\textbackslash \textbackslash postcode = '17'}} (Q1), the base data is used.

Rule 4. For a step that contains a predicate filter that evaluates a non-classified text value or contains the following or preceding axis, the base data is used. For example, \texttt{step two} (Q2) contains a predicate filter that evaluates a text value.

That completes the base rules, next we present the advanced index selection rules.

6.4.2 Advanced Index Selection Rules

Before presenting the index selection process, we must introduce the concept of a context shift.

Definition 9 A Context Shift occurs when it cannot be guaranteed that the context nodes associated with a step are within the (pre/post) range of the previous predicate filter.

For the index selection process it is crucial that the nodes that must be evaluated at the current step are within the range of the previous predicate filter (Definition 9). This is because this filter (which requires a base data evaluation) does not have to be applied to steps if it can be guaranteed that they are within the range of the previous predicate filter. A context shift can only occur after the first step to which rule 4 applies. For example, in Q2 (Figure 6.5), the predicate filter that is applied to step two is not applied to step three, but it is applied to step four. Using XPath logic, a base data evaluation is avoided at step three because it can be determined that all nodes that could possibly be evaluated at step three are within the range of the predicate filter (there is no context shift).

The following three cases result in a context shift:
Case 1. A step contains an axis in an opposite hierarchical direction to the previous step, for example, a step that contains the descendant or child axis followed by a step that contains ancestor or parent. This is exemplified in the sixth step (\texttt{/ancestor::B}) of \texttt{Q3} (Figure 6.6), that is, the fifth step contains the descendant-or-self axis and the sixth step contains the ancestor axis.

Case 2. A step that contains the \textit{preceding} or \textit{following} axis is encountered. For example, if we replaced the ancestor axis in the sixth step of \texttt{Q3} with following or preceding, the result would still require a context shift.

Case 3. Rule 4 (of the base index selection rules) applies to a step. This is exemplified in the seventh step of \texttt{Q4} as the predicate filter associated with this step contains the following axis.

In Figure 6.6, an asterisk above a step denotes a base data evaluation. When a base data evaluation is required, the unique node labels (e.g. pre/post) of the nodes must be compared. For all other steps it is sufficient to evaluate \textit{name}, \textit{class}, \textit{level}, \textit{type} and \textit{value} (i.e. the \texttt{NCLTV} index may be used instead).

Each time a context shift occurs based on case 1 or case 2, a \texttt{join} operation (that compares node labels) is performed between the step immediately preceding the step to which the context shift applies and the step that required the previous base data evaluation. For example, the join between the third and fifth steps in \texttt{Q3}. Next, the step to which the context shift applies is joined with the step to its immediate left, such as, the join between
the fifth and sixth steps in Q3.

If a context shift occurs based on case 3, the step to which this context shift applies is joined with the step that required the previous base data evaluation - the Last Node Access Step (lnaStep) variable described in the transformation process. In Q4, a context shift (based on case 3) occurs at step seven, therefore it is joined with step three. Finally, the rightmost step is always joined with the Last Node Access Step (if such a step exists).

6.5 Summary

In this section, we discuss the integrity of query results and provide a summary of the chapter.

6.5.1 Integrity Checking for Transformation Process

In §6.3, we described our Transformation Template, a construct used to construct SQL queries from XPath expressions. This process is designed to benefit from concepts such as partitions, classification of partitions and indexing low selectivity text nodes. In §6.3, we provided a detailed workflow for a sample XPath expression. Our final transformation process is a result of many iterations of converting XPath expressions to SQL expressions suitable for our indexing system. Initial transformations were fairly simple [46], different optimisations were then introduced in [47] and in [48]. At each point in the development of the transformation algorithms, we developed an integrity check to ensure that our transformations were correct.

Our early algorithms (before partitions) were based on the work of [29]. As such, we used the transformation algorithms and the processes of established researchers in the area. After we developed our own approach, we adopted a two-phase quality check. Firstly, we compared the results from each new iteration with the results from the previous iteration. Secondly, we compared
with an existing implementation (MonetDB/XQuery), by running identical queries on both systems. In this phase, we checked that: a) the node counts were identical, and (b) the name, type (element/attribute) were also identical, as the SQL query is based on ‘name’ and ‘type’. In both phases, the check was performed on all 51 queries presented in the experiments chapter.

6.5.2 Final Summary

In this chapter, we described a deployment of the BranchClassIndex within a relational database. To evaluate XPath expressions using an index that is deployed in a relational database, an XPath-to-SQL transformation process is required to transform XPath expressions to their SQL equivalents. Thus, we described the large subset of the XPath language that our system covers and described the XPath-to-SQL transformation process in detail and provided a worked example. Finally, as some text values were not included in the classification process that was described in Chapter 5, there are instances where the NCLTV covering index cannot be exploited to optimise a step. Therefore, this Chapter is completed with a description of the index selection process that determines whether the index can be used for an XPath step and when the base data must be evaluated because text values appear in the XPath expression.
Chapter 7

Experiments

This chapter provides details of the experiments used to evaluate the BranchClassIndex. In doing so, we evaluate the performance of a traditional node-based approach [31]. However, as the evidence in this evaluation shows, node-based approaches (such as [29, 31]) cannot scale to XML datasets of the size used in our study. Thus, in a separate evaluation using a relatively small dataset and queries taken from XPathMark benchmark [67], we evaluate node-based approaches at the end of the chapter in §7.5.

This chapter is structured as follows: §7.1 describes the evaluation method; in §7.2 we specify low cardinality text values empirically; in §7.3, we evaluate the real world XML case study. For reasons of experimental repeatability [44, 45], we evaluate the same XML datasets and queries as other researchers in §7.4; finally, in §7.5, we evaluate node-based approaches.

7.1 Evaluation Method

In the main evaluation, the BranchClassIndex is compared to one of the more recent path-based approaches [24], which we refer to as the PathBasedIndex. MonetDB/XQuery [12] is also evaluated because it uses the native Staircase Join algorithm [32] and because it was evaluated in [24] (the path-based
approach used in our evaluation). Finally, the commercial XML database, SQL Server 2008, is evaluated because it uses the relational XML indexing techniques described in [57], which were subsequently discussed by the authors of [31] (the node-based approach used in our evaluation).

The BranchClassIndex, the PathBasedIndex, SQL Server 2008, and MonetDB/XQuery were deployed on identical servers with a 2.66GHz Intel(R) Core 2 Duo CPU and 4GB of RAM. The BranchClassIndex and the PathBasedIndex were deployed in an Oracle 11g relational database. Oracle 11g and MonetDB/XQuery version 4.34.4 were both deployed on Fedora 12 Linux (64bit) platforms; SQL Server was deployed on a Windows 7 (64bit) platform.

For queries executed across the vendor systems, we call the count() function to ensure that any overhead associated with document reconstruction [15] is not included in the query response times. This approach was also used in [25] for evaluating the comparative query response times of vendor systems. In order to conduct a balanced evaluation, we also called the count() function on the SQL queries executed across the BranchClassIndex and the PathBasedIndex.

Each XPath query used in the experiments was run twenty one times. Using the same approach as [26, 60], the first run was ignored to ensure hot cache response times and the remaining twenty runs were averaged to provide the final result in milliseconds. A timeout of ten minutes was placed on each query to allow us to perform the evaluation in a reasonable amount of time. Each query that took longer than ten minutes is tagged as >10mins. Additionally, to calculate average query response times, each query that is tagged as >10mins is counted as 600,000 milliseconds.
7.1.1 Implementation and Deployment Decisions

We made the following practical decisions when implementing the BranchClassIndex and the PathBasedIndex:

- For performing node comparisons, *pre/post* labels are used in both the BranchClassIndex and the PathBasedIndex (both approaches are node label independent).

- Schema-oblivious [24] node storage was used in both approaches as the BranchClassIndex is a schema-oblivious approach. The authors of the PathBasedIndex described a schema-aware and a schema-oblivious version of their PathBasedIndex. We chose schema-oblivious node storage because schema-aware approaches may incur inefficiencies when wildcards exist in XPath queries, i.e. *SQL Splitting* [24] is required.

- In both approaches, attribute nodes are stored as separate nodes that are differentiated from element nodes using the *type* column. The authors of the PathBasedIndex map attribute nodes to relational columns.

- For the BranchClassIndex and the PathBasedIndex, a B-tree index was built on each column of each relation.

- Oracles's *cost based optimiser* is allowed to chose the best *query execution plan*. In other words, we did not supply *optimiser hints* (to manually control query executions plans).

The following options were selected when deploying SQL Server and MonetDB/XQuery:

- In SQL server, we built the *primary index*, and the *secondary indexes* *PATH* and *VALUE*, which are the optimisation techniques described in [57].
• In MonetDB/XQuery, the read only storage option was selected in order to exclude any performance overhead associated with updateable storage options.

7.1.2 Query Categories

The following five query categories enable us to be precise about the type of query each approach evaluates most efficiently.

QC_1. Linear Path Queries. This type of query does not contain predicate filters.

QC_2. Twig Queries without Text Values. This type of query does not evaluate text values, but contains predicate filters.

QC_3. Twig Queries with Low Selective Text Values. This type of query contains predicate filters and evaluates text values that have low selectivity.

QC_4. Twig Queries with High Selective Text Values. This type of query contains predicate filters and evaluates text values that have high selectivity.

QC_5. Single Step Path Fragment Queries. This type of query does not contain a primary path fragment (PPF), as defined for the PathBasedIndex [24], that spans more than one step in the XPath expression. This query category is used to highlight a class of XPath queries in which path-based approaches do not improve performance.

7.2 Specifying Low Selectivity Text Values

In Chapter 5, we presented two algorithms that identify low selectivity text values that are suitable for classification (i.e. suitable for indexing). The IdentifyRangeBasedCandidates algorithm identifies nodes that can have
all of their values indexed, therefore range based evaluations (e.g. year > ‘1985’ and year < ‘2000’) can be performed in the index. If a node, such as year, contains a large number of distinct text values, all of its text values may not be suitable for indexing (because of increased index size). In which case, the IdentifyEqualityBasedCandidates algorithm may identify individual text values as being suitable for indexing, but only equality evaluations (e.g. year = ‘1985’) are for such values.

Both text value identification algorithms depend on time $T$; the number of context nodes ($CN$); and the number of initial target nodes ($TN$): $T = CN \times TN$, where $T$ was determined after a number of experiments.

As a starting point, we performed a number of experiments to determine when $T$ is 1000ms (one second). For this purpose, we created temporary node repositories of various sizes and counted the maximum number of individual node comparisons that could be performed based solely on pre/post node labels in less than one second. Using the same system on which the BranchClassIndex was deployed (described earlier), we found that $T$ is 1000ms when $CN$ is 3100 and $TN$ is 3100. In other words, $(3100 \times 3100)$ is the maximum number of pre/post node comparisons that can be performed in less than one second.

However, the threshold of $N$ equal to 3100 acts only as a guideline due to the fact that, if there are a very large number of text values that occur more than $N$ times, then it can lead to a degradation in query performance. Later on, we will demonstrate the performance of the BranchClassIndex for a number of text value subsets.

### 7.3 The Bicycle Rental Dataset

A real world dataset with genuine user requirements was chosen to determine the wider impact of our work. One of our research group’s other projects includes a smart city project, which provided us with this dataset and a
Table 7.1: Bicycle Rental Queries

<table>
<thead>
<tr>
<th>No.</th>
<th>Cat</th>
<th>XPath Query</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q01</td>
<td>QC3</td>
<td>Return all information recorded for stations in Nantes. <code>&lt;bikes/city/Nantes/stations/station&gt;</code></td>
<td>1,411,451</td>
</tr>
<tr>
<td>Q02</td>
<td>QC3</td>
<td>Return the number of bikes that are free across all stations, and all days, in Dublin. <code>&lt;bikes/city/Dublin/stations/station/free&gt;</code></td>
<td>634,320</td>
</tr>
<tr>
<td>Q03</td>
<td>QC3</td>
<td>Return all stations, from all cities, that have bicycle availability information. <code>&lt;/city//stations//station/available&gt;</code></td>
<td>191,680</td>
</tr>
<tr>
<td>Q04</td>
<td>QC3</td>
<td>Return all stations, from all cities, that have weather information including: wind direction and speed; and the date and time it was received. <code>&lt;/city//stations//weather/time]/]/weather//wind/direction][/weather//wind/speed&gt;</code></td>
<td>191,680</td>
</tr>
<tr>
<td>Q05</td>
<td>QC3</td>
<td>Return all information regarding stations in Luxembourg in which there were no free bikes. <code>&lt;/Luxembourg/stations//station/available = '0'&gt;</code></td>
<td>13,046</td>
</tr>
<tr>
<td>Q06</td>
<td>QC3</td>
<td>Return the <code>id</code> (identifier) of each station that had no bikes available. <code>&lt;/stations/station//available = '0'//id&gt;</code></td>
<td>429,585</td>
</tr>
<tr>
<td>Q07</td>
<td>QC3</td>
<td>Return the cities that had a wind speed greater than 6 miles per hour. <code>&lt;/stations//wind/speed &gt; '6'//parent:*&gt;</code></td>
<td>145</td>
</tr>
<tr>
<td>Q08</td>
<td>QC5</td>
<td>Return the stations that had a wind direction of 40. <code>&lt;//direction[.= '40']//ancestor::stations//station&gt;</code></td>
<td>192,755</td>
</tr>
<tr>
<td>Q09</td>
<td>QC5</td>
<td>Return all entries for Lyon on the date 01/06/2010. <code>&lt;/Lyon[./@day = '01'][./@month = '06'][./@year = '2010']&gt;</code></td>
<td>1</td>
</tr>
<tr>
<td>Q10</td>
<td>QC5</td>
<td>Return the wind chill in Lyon on the date 01/06/2010. <code>&lt;/Lyon[./@day = '01'][./@month = '06'][./@year = '2010']//chill&gt;</code></td>
<td>642</td>
</tr>
</tbody>
</table>

Table 7.1: Bicycle Rental Queries

The BranchClassIndex does not contain text values (i.e. the branch classification process was not extended to include text values). For each of the ten queries, Table 7.4 shows the query response time for the BranchClassIndex (BCI), the PathBasedIndex (PBI), MonetDB/XQuery (MDB) and SQL Server 2008 (SQLS). For each query category, the average query response time is shown and the fastest time is highlighted with the colour blue. Additionally, the total average query response time across all query categories is shown and again the fastest time is highlighted in blue.

For the bicycle rental dataset, Table 7.5 shows: the number of nodes in the `Node` relation; the number of actual branch classes; the number of `rows` in the `NCLTV` relation; and the number of branch class relationships in the `PC_REL` and `AD_REL` relations.

1[^1]: [http://computing.dcu.ie/~isg/BicycleRental](http://computing.dcu.ie/~isg/BicycleRental)
<table>
<thead>
<tr>
<th>Query</th>
<th>Cat</th>
<th>BCI</th>
<th>PBI</th>
<th>MDB</th>
<th>SQLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q01</td>
<td>QC1</td>
<td>1,130ms</td>
<td>3,110ms</td>
<td>5,662ms</td>
<td>258ms</td>
</tr>
<tr>
<td>Q02</td>
<td>QC1</td>
<td>1,084ms</td>
<td>1,070ms</td>
<td>3,152ms</td>
<td>160ms</td>
</tr>
<tr>
<td>AVG</td>
<td></td>
<td>1,107ms</td>
<td>2,090ms</td>
<td>4,407ms</td>
<td>212ms</td>
</tr>
<tr>
<td>Q03</td>
<td>QC2</td>
<td>812ms</td>
<td>&gt;10mins</td>
<td>19,498ms</td>
<td>&gt;10mins</td>
</tr>
<tr>
<td>Q04</td>
<td>QC2</td>
<td>325ms</td>
<td>&gt;10mins</td>
<td>16,766ms</td>
<td>&gt;10mins</td>
</tr>
<tr>
<td>AVG</td>
<td></td>
<td>569ms</td>
<td>600,000ms</td>
<td>18,132ms</td>
<td>600,000ms</td>
</tr>
<tr>
<td>Q05</td>
<td>QC3</td>
<td>87,385ms</td>
<td>157,090ms</td>
<td>11,890ms</td>
<td>&gt;10mins</td>
</tr>
<tr>
<td>Q06</td>
<td>QC3</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
<td>108,192ms</td>
<td>&gt;10mins</td>
</tr>
<tr>
<td>Q07</td>
<td>QC3</td>
<td>96,143ms</td>
<td>&gt;10mins</td>
<td>107,629ms</td>
<td>&gt;10mins</td>
</tr>
<tr>
<td>AVG</td>
<td></td>
<td>261,176ms</td>
<td>452,363ms</td>
<td>75,904ms</td>
<td>600,000ms</td>
</tr>
<tr>
<td>Q09</td>
<td>QC4</td>
<td>31ms</td>
<td>140ms</td>
<td>162ms</td>
<td>1ms</td>
</tr>
<tr>
<td>Q10</td>
<td>QC4</td>
<td>484ms</td>
<td>68,690ms</td>
<td>5,215ms</td>
<td>166ms</td>
</tr>
<tr>
<td>AVG</td>
<td></td>
<td>258ms</td>
<td>34,415ms</td>
<td>2,688ms</td>
<td>84ms</td>
</tr>
<tr>
<td>Q08</td>
<td>QC5</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
<td>105,129ms</td>
<td>error</td>
</tr>
<tr>
<td>Total AVG</td>
<td></td>
<td>138,739ms</td>
<td>323,010ms</td>
<td>38,329ms</td>
<td>333,399ms</td>
</tr>
</tbody>
</table>

Table 7.2: Results for the Bicycle Rental Dataset

<table>
<thead>
<tr>
<th>NAME</th>
<th>Nodes</th>
<th>Branch Classes</th>
<th>NCLTV</th>
<th>PC_REL</th>
<th>AD_REL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BranchClassIndex</td>
<td>85,965,102</td>
<td>1,067</td>
<td>1,224</td>
<td>2,133</td>
<td>5,032</td>
</tr>
</tbody>
</table>

Table 7.3: Branch Index Statistics

Linear Path Expressions (Q01, Q02)

Queries Q01 and Q02 are linear path queries. Both SQL Server and the PathBasedIndex process all steps in linear path queries simultaneously. Conversely, the BranchClassIndex and MonetDB/XQuery perform step-at-a-times evaluations.

SQL Server (SQLS) performed best overall in this category followed by the BranchClassIndex, PathBasedIndex and MonetDB/XQuery respectively. We believe that SQL Server performed well in this category because its PATH index [20,57] is optimised for linear path queries. MonetDB/XQuery performs a Staircase Join [32] between each step in linear path expressions. The fact that SQL Server and the PathBasedIndex outperformed MonetDB/XQuery indicates that, for linear path queries, path-based approaches are more ef-
Similar to MonetDB/XQuery, the BranchClassIndex also performs a join at each step in the XPath expression. However, the size of the BranchClassIndex is small (see Table 7.5) for this dataset (in comparison to later datasets), which may be why it outperformed the PathBasedIndex. Later in this chapter, we show that (when evaluating other datasets) the PathBasedIndex can in some instances outperform the BranchClassIndex for linear path queries.

**Twig Queries without Text Values (Q03 and Q04)**

The BranchClassIndex performed best overall for Twig queries that do not evaluate text values (QC2) - see Figure 7.2. This is because the NCLTV covering index can be exploited to optimise all steps in the XPath expression (the rightmost step excepted). In contrast, the PathBasedIndex must perform a costly join between each primary path fragment (PPF). For example, query Q03 has two primary path fragments: //city//stations and /station/available, whereas query Q04 has four PPFs. One join operation is required for Q03 and four joins are required for Q04, which leads to a significant performance overhead for the PathBasedIndex. SQL Server’s PATH index suffers the same drawback. Thus, the PathBasedIndex and SQL Server show very similar query response times and they are the slowest approaches...
for queries in QC2.

MonetDB/XQuery performed better than the PathBasedIndex and SQL Server. This indicates that when node comparisons are required between path fragments, even though MonetDB/XQuery performs more joins (one between each XPath step rather than each path fragment), the Staircase Joins may perform more efficiently overall.

**Twig Queries with Low Selectivity Text Values (Q05, Q06, Q07)**

For queries in QC3, the PathBasedIndex and SQL server suffer the same drawback as they did in QC2 (joins are required between path fragments). The BranchClassIndex cannot exploit the NCLTV index at every step because queries in QC3 contain text values. Before branch classification extensions, the BranchClassIndex must access the base data to evaluate text values (because they do not exist in the index). MonetDB/XQuery performs best overall in this category indicating that, for the bicycle rental dataset, the Staircase Join is more consistent across all Twig queries. However, MonetDB/XQuery took more than 100 seconds to evaluate both Q06 and Q07.

---

2In our earlier work [46], we optimised this inefficiency associated with path-based approaches by extending path fragments to some leaf nodes within predicate filters if they have high selectivity and working backwards, thus reducing the number of nodes that must be processed in the join between path fragments. In this work, statistics including the selectivity of nodes are generated and indexed in advance of the query process.
which makes this its least efficient category.

![Twig Queries with Low Selectivity Text Values](image1.png)

Figure 7.3: Average for Twig Queries with Low Selectivity Text Values

![Twig Queries with High Selectivity Text Values](image2.png)

Figure 7.4: Average for Twig Queries with High Selectivity Text Values

Twig Queries with High Selectivity Text Values (Q09, Q10)

Even though the BranchClassIndex does not contain text values, the high selectivity of the text values in QC4 ensures that few node comparisons. The BranchClassIndex performed better than the PathBasedIndex because step two (/chill) in query Q10 requires a base data evaluation in both approaches (to locate the result nodes), but for the BranchClassIndex the query process exploits branch classes to prune the search space. The BranchClassIndex outperforms MonetDB/XQuery, indicating that it bypasses a
larger number of nodes during the query process. SQL Server performed best overall, which may be attributed to its VALUE index, which optimises text value evaluations (particularly for values that have high selectivity [20]).

**Single Step Path Fragment Queries (Q08)**

QC5 is used to illustrate a category of queries in which PathBasedIndex’s primary path fragments and SQL Server’s PATH index cannot be exploited. In other words, *root path* indexes are redundant if each path fragment (such as simple path expression in XRel [69] or primary path fragments in [24]) spans just one step each and thus no optimisation is achieved. The low selectivity of the text value ‘40’ in Q08 is responsible for the inefficiency of the BranchClassIndex. However, the PathBasedIndex performs poorly in this category even when the queries do not contain text values. This is because each primary path fragment spans just one XPath step and therefore performs at the same level as node-based approaches (node-based approaches are evaluated later in §7.5). For example, we ran the same query without the text value (\texttt{/\textit{direction}/ancestor::\textit{stations}/\textit{station}) and the PathBasedIndex took more than ten minutes to return a result, whereas the BranchClassIndex took just three seconds.

![Figure 7.5: Average Single Step Path Fragment Queries](image-url)
SQL Server does not support the ancestor axis, which is why no result is shown for Q08. MonetDB/XQuery is the most efficient approach for Q08, and combining this with the fact that it is also the most efficient for queries in QC\textsubscript{3} (Twig queries with low selectivity text values), indicates that it is the most efficient approach for Twig queries that contain low selectivity text nodes. There is overlap between QC\textsubscript{3} and QC\textsubscript{5} because Q08 can belong to either category; category QC\textsubscript{5} is only used to illustrate a category of queries that path-based approaches cannot optimise.

**Overall Performance**

Across all queries, the average performance of each approach is illustrated in Figure 7.6. These averages show that MonetDB/XQuery is the most highly optimised approach overall, followed by the BranchClassIndex, the PathBasedIndex and SQL Server respectively. However, we now show how the BranchClassIndex can be optimised further by classifying text values.

![Overall Performance](image)

**Figure 7.6: Average Performance across all Query Categories**
<table>
<thead>
<tr>
<th>Query</th>
<th>Cat</th>
<th>BCI_1</th>
<th>BCI_2</th>
<th>BCI_3</th>
<th>BCI_4</th>
<th>BCI_5</th>
<th>BCI_6</th>
<th>PBI</th>
<th>MDB</th>
<th>SQLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q01</td>
<td>QC1</td>
<td>1,130ms</td>
<td>1,120ms</td>
<td>5,718ms</td>
<td>12,288ms</td>
<td>1,041ms</td>
<td>4,104</td>
<td>3,110ms</td>
<td>5,662ms</td>
<td>258ms</td>
</tr>
<tr>
<td>Q02</td>
<td>QC2</td>
<td>1,084ms</td>
<td>1,083ms</td>
<td>3,913ms</td>
<td>7,590ms</td>
<td>1,660ms</td>
<td>2,159</td>
<td>1,070ms</td>
<td>3,152ms</td>
<td>166ms</td>
</tr>
<tr>
<td>AVG</td>
<td></td>
<td>1,107ms</td>
<td>1,102ms</td>
<td>4,816ms</td>
<td>9,939ms</td>
<td>1,354ms</td>
<td>3,132ms</td>
<td>2,090ms</td>
<td>4,407ms</td>
<td>212ms</td>
</tr>
<tr>
<td>Q03</td>
<td>QC2</td>
<td>812ms</td>
<td>103ms</td>
<td>12,249ms</td>
<td>13,582ms</td>
<td>269ms</td>
<td>317</td>
<td>&gt;10mins</td>
<td>19,498ms</td>
<td>&gt;10mins</td>
</tr>
<tr>
<td>Q04</td>
<td>QC2</td>
<td>325ms</td>
<td>405ms</td>
<td>72,372ms</td>
<td>70,932ms</td>
<td>2,078</td>
<td>3,046</td>
<td>&gt;10mins</td>
<td>16,766ms</td>
<td>&gt;10mins</td>
</tr>
<tr>
<td>AVG</td>
<td></td>
<td>569ms</td>
<td>254ms</td>
<td>42,311ms</td>
<td>42,257ms</td>
<td>1,174ms</td>
<td>1,682ms</td>
<td>600,000ms</td>
<td>18,132ms</td>
<td>600,000ms</td>
</tr>
<tr>
<td>Q05</td>
<td>QC3</td>
<td>87,350ms</td>
<td>87,972</td>
<td>2,002ms</td>
<td>2,404ms</td>
<td>157ms</td>
<td>245</td>
<td>157,000ms</td>
<td>11,890ms</td>
<td>&gt;10mins</td>
</tr>
<tr>
<td>Q06</td>
<td>QC3</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
<td>14,479ms</td>
<td>8,232ms</td>
<td>1,180ms</td>
<td>1,241</td>
<td>&gt;10mins</td>
<td>108,192ms</td>
<td>&gt;10mins</td>
</tr>
<tr>
<td>Q07</td>
<td>QC3</td>
<td>96,143ms</td>
<td>96,143</td>
<td>1,433ms</td>
<td>1,779ms</td>
<td>163ms</td>
<td>102</td>
<td>&gt;10mins</td>
<td>107,629ms</td>
<td>&gt;10mins</td>
</tr>
<tr>
<td>AVG</td>
<td></td>
<td>261,176ms</td>
<td>261,371ms</td>
<td>5,971ms</td>
<td>4,138ms</td>
<td>466ms</td>
<td>529ms</td>
<td>452,363ms</td>
<td>75,904ms</td>
<td>600,000ms</td>
</tr>
<tr>
<td>Q09</td>
<td>QC4</td>
<td>31ms</td>
<td>147ms</td>
<td>197ms</td>
<td>156ms</td>
<td>8ms</td>
<td>369</td>
<td>140ms</td>
<td>162ms</td>
<td>1ms</td>
</tr>
<tr>
<td>Q10</td>
<td>QC4</td>
<td>484ms</td>
<td>614ms</td>
<td>11,658</td>
<td>12,056</td>
<td>7,957ms</td>
<td>7,026</td>
<td>68,690ms</td>
<td>5,215ms</td>
<td>166ms</td>
</tr>
<tr>
<td>AVG</td>
<td></td>
<td>258ms</td>
<td>381ms</td>
<td>5,928ms</td>
<td>6,106ms</td>
<td>3,983ms</td>
<td>3,698ms</td>
<td>34,415ms</td>
<td>2,688ms</td>
<td>84ms</td>
</tr>
<tr>
<td>Q08</td>
<td>QC5</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
<td>1,388ms</td>
<td>1,391ms</td>
<td>2,043ms</td>
<td>2,175ms</td>
<td>&gt;10mins</td>
<td>105,129ms</td>
<td>error</td>
</tr>
<tr>
<td>Total AVG</td>
<td></td>
<td>138,739ms</td>
<td>138,758ms</td>
<td>12,540ms</td>
<td>13,041ms</td>
<td>1,646ms</td>
<td>2,078ms</td>
<td>323,010ms</td>
<td>38,329ms</td>
<td>333,399ms</td>
</tr>
</tbody>
</table>

Table 7.4: Bicycle Rental Dataset Results (after Text Value Classification)
7.3.2 Query Analysis after Text Value Classifications

The BranchClassIndex is now evaluated under a number of separate conditions to highlight the benefit of its individual components. The query response times are shown before and after the classification process has been extended to include text values. In addition, the query response times are shown with and without the inclusion of the Parent and Child ClassChain components, which (as discussed in Chapter 5) are optional.

For the bicycle rental dataset, the text value identification algorithms identified a total of 519 low selectivity text values as being suitable for classification. Of these, we classified a large subset (227) for the experiments described in this section (this subset includes the low selectivity text values in query categories QC3 and QC5). The following six versions of the BranchClassIndexes are now evaluated:

**BCI.1.** The BranchClassIndex before classification of text values and including Parent and Child ClassChain Components.

**BCI.2.** The BranchClassIndex before classification of text values and *not* Parent and Child ClassChain Components.

**BCI.3.** The BranchClassIndex with 227 text values and including the Parent and Child ClassChain Components.

**BCI.4.** The BranchClassIndex with 227 text values and *not* including the Parent and Child ClassChain Components.

**BCI.5.** The BranchClassIndex with 100 text values and including the Parent and Child ClassChain Components.

**BCI.6.** The BranchClassIndex with 100 text values and *not* including Parent and Child ClassChain Components.
Twig Queries with Low Selectivity Text Values (Q05, Q06, Q07)

The performance value of branch classification extensions is clear for queries that evaluate text values that are of low selectivity (QC3). Figure 7.7 shows that the BranchClassIndex performs significantly better when text values are indexed (BCI3, BCI4, BCI5, BCI6) than when they are not (BCI1, BCI2). Even when a large subset (227) of text values are classified (BCI3, BCI4) significant performance gains are achieved. However, when fewer text values are classified (100), the BranchClassIndex performs better.

In our earlier assessment of queries in category QC3, MonetDB/XQuery performed best overall and queries that evaluate text values that are of low selectivity...
selectivity were identified as worst case queries for the BranchClassIndex. In contrast, after classification extensions, the BranchClassIndex is the most highly optimised approach even when a large subset of text values are classified (see Figure 7.7).

**Linear Path Expressions (Q01, Q02)**

Linear path queries only evaluate the base data at the rightmost step (to locate the result set). For this reason, extending the classification process to include text values cannot optimise linear path queries. In fact, they can incur a performance overhead. This is illustrated in Figure 7.8, which shows that BranchClassIndex before the inclusion of text values (BCI1 and BCI2) performs better than after their inclusion (BCI3, BCI4, BCI5, BCI6).

Additionally, Figure 7.8 shows that when the classification process includes 100 text values (BCI3, BCI4) it performs better than when it contains 227 text values (BCI5, BCI6). This is because the number of branch classes increases, and in particular, the PC_REL and AD_REL relations grow significantly in size. There is therefore a trade off between the number of text values that are included in the classification process and the performance overhead they incur for linear path queries. However, a subset of 100 text values
values (BCI\textsubscript{5}) performs almost as well as the BranchClassIndex before the extensions.

Note also that the optional Parent and Child ClassChain components minimise the effect of the increased index size (e.g. BCI\textsubscript{3} performs better than BCI\textsubscript{4} and BCI\textsubscript{5} performs better than BCI\textsubscript{6}), whereas when text values are not classified these components do not have a query performance benefit (BCI\textsubscript{1} and BCI\textsubscript{2} perform equally well). The reason for this is that the Parent and Child ClassChain components are stored in the PC\_REL relation. An examination of the size of the PC\_REL relation in contrast to the AD\_REL relation (Figure 7.5) shows a much larger differentiation when text values are classified. For example, for BCI\textsubscript{1} the AD\_REL relation contains around 3000 more relationships than the AD\_REL relation. In contrast, for BCI\textsubscript{3} the differential is around 7 million relationships, thus the Parent and Child ClassChain components have a significant search space pruning effect.

**Twig Queries without Text Values (Q03, Q04)**

Figure 7.9 shows that the results for Twig queries that do not contain text values are similar to those for linear path queries. Again, the reason for this is that by specification, queries in QC\textsubscript{2} do not evaluate text values. Therefore, the inclusion of text values in the index has no value and may incur a performance overhead due to the increased index size.

The only difference shown between queries in QC\textsubscript{1} and those in QC\textsubscript{2} is that the benefit of the Parent and Child ClassChain components in QC\textsubscript{2} is not significant. The reason for this is that in total, the queries in QC\textsubscript{1} contain eleven parent or child axes, whereas those in QC\textsubscript{2} contain only five. In other words, the particular queries in QC\textsubscript{2} cannot exploit the Parent and Child ClassChain components as often as those in QC\textsubscript{1}.
Twig Queries with High Selectivity Text Values (Q09, Q10)

In category QC4, the high selectivity of the text nodes ensures that, even though the base data must be evaluated (to access the text values), the text values have high selectivity leading to few node comparisons. For queries in QC4, the BranchClassIndex performed better than the PathBasedIndex and MonetDB/XQuery, even after the classification of text values (see Figure 7.10).
Single Step Path Fragment Queries

For single path fragment queries, the BranchClassIndex is optimised because the low cardinality text value (‘4θ’) in query Q08 was classified. Again, MonetDB/XQuery was the most efficient approach in this category in the earlier evaluation, but now the BranchClassIndex is more efficient (BCL_3, BCL_4, BCL_5, BCL_6).

![Single Step Path Fragment Queries](image)

Figure 7.11: Average Single Step Path Fragment Queries

7.3.3 Overall Query Performance

In Table 7.4 the average (AVG) query response time (milliseconds) is shown for each approach. The bar chart in Figure 7.12 illustrates these average times. The BranchClassIndex now performs best overall when a large (BCL_3, BCL_4) or a small (BCL_5, BCL_6) subset of low selectivity text values are indexed.

7.4 Comparison Using Standard Benchmarks

In this section, we evaluate the performance of each approach using the XPathMark benchmark [67], the Computer Science Bibliography [21] and the Protein Sequence Database [58]. In the following experiments, the
BranchClassIndex is evaluated (1) without text values and including the Parent and Child ClassChain components (BCL_1), (2) without text values and not including the Parent and Child ClassChain components (BCL_2), and (3) including all of the text values identified as suitable for classification (BCL_2).

No. | Cat | XPath Query                                                                 | Nodes |
--- | --- |------------------------------------------------------------------------------|-------|
Q13 | QC1 | /site/regions/africa                                                         | 1     |
Q14 | QC1 | /site/people/person[@id = 'person0']                                       | 1     |
Q15 | QC1 | //regions/africa//item/name                                                  | 6,600 |
Q16 | QC1 | //person/profile/[income]/name                                              | 151,539|
Q17 | QC1 | //people/person/profile/[gender][profile/age]/name                          | 38,583|
Q18 | QC1 | //keyword/ancestor::listitem/text/keyword                                   | 373,260|
Q19 | QC1 | /site/closed_auctions/closed_auction//keyword                               | 15,004|
Q20 | QC1 | /site/closed_auctions/closed_auction//descendant::keyword//date              | 64,133|
Q21 | QC1 | /site/closed_auctions/closed_auction//annotation//description//text/keyword | 48,632|
Q22 | QC1 | /site/closed_auctions/closed_auction//annotation//description//text//keyword  | 31,773|
Q23 | QC1 | /site//closed_auction/annotation//text//keyword//date                         | 64,133|
Q24 | QC1 | /site/item//description//listitem//mailbox//text                             | 75,940|
Q25 | QC1 | /site/item//description//listitem//mailbox//text                             | 72,748|

Table 7.6: XMark Queries

7.4.1 The XPathMark Benchmark

The XPathMark benchmark has been widely used by the research community to evaluate the performance of XPath processors [2, 9, 24, 54]. The XMark [67] dataset is used in XPathMark and because it contains synthetic
Table 7.7: Results for the Mark Queries

<table>
<thead>
<tr>
<th>Query</th>
<th>Cat</th>
<th>BCI_1</th>
<th>BCI_2</th>
<th>BCI_3</th>
<th>PBI</th>
<th>MDB</th>
<th>SQLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q13</td>
<td>QC_1</td>
<td>3ms</td>
<td>2ms</td>
<td>12ms</td>
<td>12ns</td>
<td>120ns</td>
<td>1ms</td>
</tr>
<tr>
<td>Q15</td>
<td>QC_3</td>
<td>470ms</td>
<td>3.628ms</td>
<td>713ms</td>
<td>21ms</td>
<td>128ns</td>
<td>147ns</td>
</tr>
<tr>
<td>Q18</td>
<td>QC_1</td>
<td>3,144ms</td>
<td>4.431ms</td>
<td>8.925ms</td>
<td>&gt;10mins</td>
<td>9,620ns</td>
<td>error</td>
</tr>
<tr>
<td>Q19</td>
<td>QC_3</td>
<td>322ms</td>
<td>769ms</td>
<td>464ms</td>
<td>153ms</td>
<td>622ns</td>
<td>582ms</td>
</tr>
<tr>
<td>Q21</td>
<td>QC_1</td>
<td>548ms</td>
<td>1,590ms</td>
<td>829ms</td>
<td>47ms</td>
<td>623ns</td>
<td>10ms</td>
</tr>
</tbody>
</table>

**AVG**

| Q16   | QC_1| 1,907ms| 1.885ms| 3.879ms| >10mins| 1.162ms| 1.235ms|
| Q17   | QC_1| 3,389ms| 11.762ms| 60.573ms| >10mins| 1.131ms| 973ns |
| Q20   | QC_3| 4,018ms| 8.797ms| 6.414ms| >10mins| 649ms| 2.754ms|
| Q22   | QC_3| 2,706ms| 22.328ms| 3.880ms| >10mins| 663ms| 271ms |
| Q23   | QC_3| 1,419ms| 2.603ms| 3.991ms| >10mins| 672ms| >10mins|
| Q24   | QC_3| 1,889ms| 1.873ms| 3.311ms| >10mins| 1.516ms| >10mins|
| Q25   | QC_3| 4,291ms| 4.259ms| 7.880ms| >10mins| 1.559ms| >10mins|

**AVG**

| Q14   | QC_1| 168ms  | 159ms  | 357ms  | 235ms | 134ns| 118ms |

**Total AVG**

| Q14   | QC_1| 1,798ms| 4,907ms| 7,787ms| 369,266ms| 1,430ms| 150,507ms|

In Table 7.6, queries Q15-Q22 are taken directly from XPathMark; queries Q13 and Q14 are taken from [31] (which is the node-based approach we evaluate later in §7.5); and finally, we added queries Q23, Q24, and Q25 to illustrate some issues. Within the XMark dataset, a total of 132 text values where identified as suitable for classification all of which are indexed in BCI_3.

For linear path queries (QC_1), SQL Server performed best overall (see Table 7.13). We attribute this to SQL’s PATH index which is highly optimised for non-branching queries. The BranchClassIndex outperformed the Path-BasedIndex for linear path queries because query Q18 took the PathBasedIndex more than ten minutes. The reason for this is that the ancestor axis in step three splits the query into three separate primary path fragments and inefficient node comparisons are required between each pair. This shows that the PathBasedIndex can perform poorly for queries in category QC_1 that consist of more than one primary path fragment (an issue that was not
highlighted in the bicycle rental experiments). The BranchClassIndex can always exploit the NCLTV covering index for all but the rightmost step of any linear path query, and therefore outperforms the PathBasedIndex. As with the bicycle rental dataset, the BranchBasedIndex outperforms MonetDB/XQuery for linear path queries, which suggests that it bypasses a larger number of nodes during query processing.

<table>
<thead>
<tr>
<th>NAME</th>
<th>Nodes</th>
<th>Branch Classes</th>
<th>NCLTV</th>
<th>PC_REL</th>
<th>AD_REL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCL1/2</td>
<td>24,645,234</td>
<td>1,337,193</td>
<td>1,653,814</td>
<td>2,676,477</td>
<td>7,459,066</td>
</tr>
<tr>
<td>BCL_3</td>
<td>24,645,234</td>
<td>3,097,821</td>
<td>3,817,329</td>
<td>6,198,038</td>
<td>14,403,306</td>
</tr>
</tbody>
</table>

Table 7.8: BranchClassIndex Statistics (XMark)

MonetDB/XQuery performed better than the BranchClassIndex for Twig queries that do not evaluate text values QC₂, whereas in the bicycle rental evaluation, the BranchClassIndex was the most efficient approach. We believe that the reason for this is that, for the XMark dataset, the BranchClassIndex contains a larger number of relationships (see Table 7.8) than it did for the bicycle rental dataset (see Table 7.5). The BranchClassIndex is the second most efficient approach. The PathBasedIndex does not perform well for Twig queries (unless they have high selectivity). This is because costly joins are performed between each branching path (primary path fragment).

SQL Server also did not perform well overall in category QC₂, but it did perform well in queries Q16, Q17, Q20 and Q22. We added queries Q23, Q24, and Q25 (in addition to those in the XPathMark benchmark) to highlight a limitation of SQL Server. Notice that Q23 is a slight variation of Q22 (the predicate contains multiple ancestor-descendant relationships rather than parent-child relationships). However, the query response time for Q22 is 271ms, whereas Q23 took longer than ten minutes. This suggests that SQL Server’s PATH index is efficient for paths that contain multiple parent-child relationships, but not for ancestor-descendant relationships. Queries Q24
and Q25 illustrate this point further.

For Q14 (the only query in QC₄), all approaches took less than 1 second to return the result. This shows again that all approaches performed well for Twig queries that have high selectivity (QC₁). Across all query categories, MonetDB/XQuery is the most efficient approach, followed closely by the BranchClassIndex (368ms less efficient overall). SQL Server and the PathBasedIndex are some orders of magnitude less efficient than MonetDB/XQuery and the BranchClassIndex. Finally, SQL Server is more than twice as efficient as the PathBasedIndex.

**BranchClassIndex Evaluation**

Indexing text values incurs an overhead for queries (BCI₃ is four time slower overall than BCI₁). The reason for this is the increased index traversal costs for BCI₃ as the index itself is larger than BCI₂ (see Table 7.8).

The inclusion of the Parent and Child ClassChain components has a clear benefit (BCI₁ is almost three times more efficient then BCI₂). This is because the PC.REL relation contains significantly fewer relationships than the AD.REL relation (see Table 7.8). One factor that increases the number of ancestor/descendant relationships is the *nested-depth*. For example, the XMark dataset has 13 levels, whereas the bicycle rental has only 8. Another factor that can impact the number of ancestor/descendant relationships is text value classification (as pointed out in the bicycle rental evaluation).

**7.4.2 The Computer Science Bibliography**

Queries executed across the Computer Science Bibliography are mostly taken from other published works in the area of XPath query optimisation [4, 13, 46, 47]. SQL Server 2008 is not included in this evaluation because it had difficulty loading an XML file that has an associated DTD (Document Type Descriptor). A total of 80 text values were identified as suitable for classifi-
Table 7.9: Computer Science Bibliography Queries

<table>
<thead>
<tr>
<th>Query</th>
<th>Cat</th>
<th>XPath Query</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q26</td>
<td>QC3</td>
<td>/dblp/article[@year='1991']/*/key</td>
<td>8,126</td>
</tr>
<tr>
<td>Q27</td>
<td>QC1</td>
<td>/dblp/article/author</td>
<td>1,235,495</td>
</tr>
<tr>
<td>Q28</td>
<td>QC1</td>
<td>/dblp/article/sub</td>
<td>3,868</td>
</tr>
<tr>
<td>Q29</td>
<td>QC1</td>
<td>/dblp/article/title[@sub='2']</td>
<td>590</td>
</tr>
<tr>
<td>Q30</td>
<td>QC1</td>
<td>/dblp/inproceedings/*/booktitle</td>
<td>805,306</td>
</tr>
<tr>
<td>Q31</td>
<td>QC6</td>
<td>/sub/ancestor::inproceedings</td>
<td>698</td>
</tr>
<tr>
<td>Q32</td>
<td>QC2</td>
<td>/dblp/inproceedings/title[@/i]/sub</td>
<td>176</td>
</tr>
<tr>
<td>Q33</td>
<td>QC4</td>
<td>/dblp/inproceedings[@title='Semantic Analysis Patterns.']/author</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 7.10: Results for the Computer Science Bibliography

For linear path queries (QC1), the BranchClassIndex performed best overall, followed by the PathBasedIndex and then MonetDB/XQuery (see Table 7.10). Thus, again in this category, MonetDB/XQuery’s Staircase join performs poorly when compared to approaches that exploit structured information for query processing.

For Twig queries without text values (QC2), the BranchClassIndex performs best overall followed by MonetDB/XQuery and then the PathBasedIndex.

Table 7.11: Branch Index Statistics (DBLP)
The PathBasedIndex performed three (costly) joins as Q32 (the only query in this category) consists of three primary path fragments. It is our understanding that the join between steps two and three has a significant performance overhead for MonetDB/XQuery as there are a large number of inproceedings and title nodes in the computer science bibliography.

Queries in category QC₃ are worst case queries for the BranchClassIndex when text values are not classified. The benefit of classifying text values that have low selectivity is clear as BCI₃ performs 2,079 times faster than BCI₁ and BCI₂. In addition, for the computer science bibliography, there is little performance overhead associated with classifying text values. This is because, in spite of the fact that there is an increase in the size of the BranchClassIndex after the inclusion of text values (see BCI₃ Table 7.11), there are still fewer than 300,000 relationships in the PC_REL and AD_REL relations.

Queries that have high selectivity (QC₄) show some interesting results. The BranchClassIndex outperforms MonetDB/XQuery and the PathBasedIndex considerably. However, we expected much better results from both these approaches as queries in QC₄ are predicated in text values that have high selectivity. Thus, join ordering or poor query execution plans may be responsible for these inefficiencies.

The only query in QC₅ is Q31. The BranchClassIndex is the most efficient approach (both before and after text values are indexed) followed by MonetDB/XQuery. In Q31, the ancestor axis at step two ensures that each primary path fragment for the PathBranchIndex approach only spans a single step. Therefore, the PathBaseIndex reduces to the performance of a node-based approach such as the XPath Accelerator.

The BranchClassIndex performs best overall across all categories when the index contains all identified text values and for the computer science bibliography, there is little overhead associated with including the text values.
across all categories. MonetDB/XQuery is second followed by the Path-BasedIndex. The Parent and Child ClassChain components do not show a performance benefit for this dataset.

<table>
<thead>
<tr>
<th>No.</th>
<th>Cat</th>
<th>XPath Query</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q34</td>
<td>QC1</td>
<td>//reference/refinfo//author</td>
<td>5,668,287</td>
</tr>
<tr>
<td>Q35</td>
<td>QC1</td>
<td>//ProteinEntry//accinfo/xrefs</td>
<td>281,246</td>
</tr>
<tr>
<td>Q36</td>
<td>QC1</td>
<td>//ProteinDatabase/ProteinEntry[/protein//name</td>
<td>262,525</td>
</tr>
<tr>
<td>Q37</td>
<td>QC1</td>
<td>//ProteinDatabase/ProteinEntry[/protein//alt-name</td>
<td>42,615</td>
</tr>
<tr>
<td>Q38</td>
<td>QC1</td>
<td>//ProteinDatabase/ProteinEntry//year/ancestor:refinfo</td>
<td>314,763</td>
</tr>
<tr>
<td>Q39</td>
<td>QC1</td>
<td>//ProteinDatabase/ProteinEntry//reference//refinfo//year</td>
<td>314,763</td>
</tr>
<tr>
<td>Q40</td>
<td>QC1</td>
<td>//ProteinDatabase/ProteinEntry/year/ancestor:reference</td>
<td>314,763</td>
</tr>
<tr>
<td>Q41</td>
<td>QC1</td>
<td>//ProteinDatabase/ProteinEntry//classification//superfamily</td>
<td>186,700</td>
</tr>
<tr>
<td>Q42</td>
<td>QC1</td>
<td>//ProteinDatabase/ProteinEntry//reference/refinfo/authors/author</td>
<td>5,668,247</td>
</tr>
<tr>
<td>Q43</td>
<td>QC1</td>
<td>//ProteinDatabase/ProteinEntry//reference/refinfo/authors/author</td>
<td>5,668,247</td>
</tr>
<tr>
<td>Q44</td>
<td>QC2</td>
<td>//ProteinDatabase/ProteinEntry//accession</td>
<td>1</td>
</tr>
<tr>
<td>Q45</td>
<td>QC4</td>
<td>//ProteinDatabase/ProteinEntry//accession</td>
<td>404</td>
</tr>
<tr>
<td>Q46</td>
<td>QC4</td>
<td>//ProteinDatabase/ProteinEntry//accession</td>
<td>258,216</td>
</tr>
<tr>
<td>Q47</td>
<td>QC4</td>
<td>//ProteinDatabase/ProteinEntry//accession</td>
<td>7</td>
</tr>
<tr>
<td>Q48</td>
<td>QC2</td>
<td>//ProteinDatabase/ProteinEntry//accession</td>
<td>9,177</td>
</tr>
<tr>
<td>Q49</td>
<td>QC2</td>
<td>//ProteinDatabase/ProteinEntry//accession</td>
<td>32,107</td>
</tr>
<tr>
<td>Q50</td>
<td>QC2</td>
<td>//ProteinDatabase/ProteinEntry//accession</td>
<td>27,429</td>
</tr>
<tr>
<td>Q51</td>
<td>QC3</td>
<td>//ProteinDatabase/ProteinEntry//accession</td>
<td>1.477</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.12: Protein Sequence Queries

7.4.3 The Protein Sequence Database

The queries executed across the Protein Sequence Database are largely taken from the experiments presented in [25]. The text node identification algorithms identified a total of 243 text values as being suitable for classification, all of which are indexed in BCI_3.

For linear path queries (QC_1), SQL Server performs marginally better than the BranchClassIndex and MonetDB/XQuery respectively. The PathBasedIndex took more than 10 minutes to evaluate queries Q38 and Q40. This is because they both contain the ancestor axis, which splits each of these queries into two primary path fragments and costly node comparisons are required between each pair. As with the bicycle rental and XMark datasets, the classification of text values has a performance overhead for linear path queries (BCI_1 performs better than BCI_3).

For Twig queries that do not contain text values (QC_2), the BranchClassIndex performs best overall because all but the rightmost step can be evaluated.
Table 7.13: Results for the Protein Sequence Database

<table>
<thead>
<tr>
<th>Query</th>
<th>Cat</th>
<th>BCI_1</th>
<th>BCI_2</th>
<th>BCI_3</th>
<th>PBI</th>
<th>MDB</th>
<th>SQLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q34</td>
<td>QC1</td>
<td>3.232ms</td>
<td>3.382ms</td>
<td>20.852ms</td>
<td>6.274ms</td>
<td>3.945ms</td>
<td>14.926ms</td>
</tr>
<tr>
<td>Q35</td>
<td>QC1</td>
<td>1.896ms</td>
<td>2.262ms</td>
<td>8.590ms</td>
<td>230ms</td>
<td>3.824ms</td>
<td>485ms</td>
</tr>
<tr>
<td>Q36</td>
<td>QC1</td>
<td>1.360ms</td>
<td>1.268ms</td>
<td>11.769ms</td>
<td>223ms</td>
<td>3.780ms</td>
<td>658ms</td>
</tr>
<tr>
<td>Q37</td>
<td>QC1</td>
<td>1.150ms</td>
<td>1.216ms</td>
<td>5.491ms</td>
<td>72ms</td>
<td>3.755ms</td>
<td>193ms</td>
</tr>
<tr>
<td>Q38</td>
<td>QC1</td>
<td>2.196ms</td>
<td>2.276ms</td>
<td>14.215ms</td>
<td>&gt;10mins</td>
<td>4.095ms</td>
<td>error</td>
</tr>
<tr>
<td>Q39</td>
<td>QC1</td>
<td>3.925ms</td>
<td>3.882ms</td>
<td>29.520ms</td>
<td>261ms</td>
<td>3.877ms</td>
<td>618ms</td>
</tr>
<tr>
<td>Q40</td>
<td>QC1</td>
<td>2.066ms</td>
<td>2.100ms</td>
<td>10.100ms</td>
<td>&gt;10mins</td>
<td>4.111ms</td>
<td>error</td>
</tr>
<tr>
<td>Q41</td>
<td>QC1</td>
<td>1.072ms</td>
<td>1.118ms</td>
<td>4.250ms</td>
<td>159ms</td>
<td>3.789ms</td>
<td>344ms</td>
</tr>
<tr>
<td>Q42</td>
<td>QC1</td>
<td>14.215ms</td>
<td>13.320ms</td>
<td>255.703ms</td>
<td>&gt;10mins</td>
<td>4.286ms</td>
<td>error</td>
</tr>
<tr>
<td>AVG</td>
<td>QC1</td>
<td>3.445ms</td>
<td>3.425ms</td>
<td>40.057ms</td>
<td>134.843ms</td>
<td>3.944ms</td>
<td>3.080ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query</th>
<th>Cat</th>
<th>BCI_2</th>
<th>BCI_3</th>
<th>PBI</th>
<th>MDB</th>
<th>SQLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q44</td>
<td>QC2</td>
<td>1.751ms</td>
<td>2.651ms</td>
<td>18.112ms</td>
<td>&gt;10mins</td>
<td>3.942ms</td>
</tr>
<tr>
<td>Q46</td>
<td>QC2</td>
<td>5.557ms</td>
<td>7.156ms</td>
<td>117.920ms</td>
<td>&gt;10mins</td>
<td>5.613ms</td>
</tr>
<tr>
<td>Q48</td>
<td>QC2</td>
<td>996ms</td>
<td>1.800ms</td>
<td>1.362ms</td>
<td>&gt;10mins</td>
<td>4.057ms</td>
</tr>
<tr>
<td>Q50</td>
<td>QC2</td>
<td>1.494ms</td>
<td>1.406ms</td>
<td>3.806ms</td>
<td>&gt;10mins</td>
<td>4.087ms</td>
</tr>
<tr>
<td>Q52</td>
<td>QC2</td>
<td>1.170ms</td>
<td>1.220ms</td>
<td>4.990ms</td>
<td>&gt;10mins</td>
<td>3.959ms</td>
</tr>
<tr>
<td>AVG</td>
<td>QC2</td>
<td>2.794ms</td>
<td>2.739ms</td>
<td>29.250ms</td>
<td>600.000ms</td>
<td>4.332ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query</th>
<th>Cat</th>
<th>BCI_3</th>
<th>PBI</th>
<th>MDB</th>
<th>SQLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q51</td>
<td>QC3</td>
<td>12.102ms</td>
<td>22.525ms</td>
<td>4.054ms</td>
<td>&gt;10mins</td>
</tr>
<tr>
<td>AVG</td>
<td>QC3</td>
<td>114ms</td>
<td>117ms</td>
<td>793ms</td>
<td>4.902ms</td>
</tr>
</tbody>
</table>

Total AVG: 10,429ms, 11,799ms, 15,833ms, 301,334ms, 5,507ms, 176,429ms

Table 7.14: Branch Index Statistics (Protein)

<table>
<thead>
<tr>
<th>NAME</th>
<th>Nodes</th>
<th>Branch Classes</th>
<th>NCLTV</th>
<th>PC_REL</th>
<th>AD_REL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCI_1/2</td>
<td>22,358,584</td>
<td>2,352,767</td>
<td>2,648,893</td>
<td>4,667,447</td>
<td>8,309,456</td>
</tr>
<tr>
<td>BCI_3</td>
<td>22,358,584</td>
<td>5,476,253</td>
<td>6,071,961</td>
<td>10,867,244</td>
<td>21,003,496</td>
</tr>
</tbody>
</table>

in NCLTV index. The PathBasedIndex and SQL Server have to perform a join at each path fragment and do not perform well in this category because there is at least one path fragment for each branching path (i.e. predicate filter) in the query. MonetDB/XQuery shows the second best performance in this category. This again indicates that when the PathBasedIndex or SQL Server must perform joins, MonetDB/XQuery’s Staircase Join will perform them more efficiently even though it may perform more of them (one at each step rather than one between each path fragment pair).

Query Q51 is only one query that uses low selectivity text values (QC3). For the first time, the BranchClassIndex performed best in this category (before text value classification); MonetDB/XQuery performed best in all previous
cases. We believe that there are two reasons for this. Firstly, even though the text value ‘1998’ is not indexed, it only occurs 25,773 times, which limits the number of node comparisons. Secondly, there are nine steps in Q51 (the most out of all queries considered here) and MonetDB/XQuery must perform a Staircase join between each pair. As previously, the PathBasedIndex and SQL Server perform at the same level in this category because of the large number of path fragments. However, query Q51 is three times faster after text values have been classified (BCI 3). The Parent and Child ClassChain components result in increased performance for query Q51 because of the large number of child axes and the fact that the PC_REL relation contains half as many relationships as the AD_REL relation (see Table 7.14).

For queries that evaluate high selectivity text values (QC 4), SQL Server took more than ten minutes to return a result for Q43 and Q47. On all previous experiments, SQL Server performed well in this category. The reason for this is unclear to us, but it may be related to join ordering. The BranchClassIndex performed best overall, which suggests that the query optimiser chooses better execution plans than it does for the PathBasedIndex. For queries with high selectivity, the classification of text nodes and the inclusion of the Parent and Child ClassChain components show little benefit (BCI 1, BCI 2, and BCI 3 perform at the same level).

Overall MonetDB/XQuery performs best, followed by the BranchClassIndex (before text values are indexed (BCI 1)). This suggests that the inclusion of all 243 text values (BCI 3) is not optimal. However, the classification of text values significantly increases the performance of queries in QC 3 (Q51 was reduced from 12 seconds to 4 seconds). Comparatively, SQL Server and the PathBasedIndex are inefficient for the protein sequence dataset.
7.5 Node Based Approaches

In this section, we will evaluate the performance of a traditional node-based approach to XPath optimisation [31], which is an optimised version of the XPath accelerator [29]. Additionally, an evaluation of the node partitioning approach that is most similar to ours [43] is provided. We will refer to [31] as the NodeApproach and [43] as the PartitionApproach.

For the NodeApproach, we built the following three partitioned B-tree indexes as suggested in [31]: \(\text{Node}(\text{level}, \text{pre})\), \(\text{Node}(\text{type}, \text{name}, \text{pre})\) and \(\text{Node}(\text{type}, \text{name}, \text{level}, \text{pre})\). Additionally we built indexes on \(\text{size}\), \(\text{name}\), \(\text{level}\), \(\text{value}\) and \(\text{type}\). For the PartitionApproach, we used partitioning factors 20, 40, 60, and 100. As suggested in [43], \(\text{Node}(\text{pre})\) is a primary key; \(\text{Node}(\text{part})\) is a foreign key reference to the primary key \(\text{Part}(\text{part})\); and indexes were built on \(\text{Node}(\text{post})\), \(\text{Node}(\text{name})\), \(\text{Node}(\text{part})\), \(\text{Part}(\text{pre})\), and \(\text{Part}(\text{post})\).

Our overall finding for both approaches is that they do not scale well, even for relatively small XML documents. As such, we had to evaluate these approaches using a relatively small dataset. Thus, for the following experiments, we generated an XMark dataset of just 115 MB in size and tested both approaches against queries from the XPathMark [67] benchmark and some queries taken from [31] (these queries are shown in Table 7.6).

<table>
<thead>
<tr>
<th>Query</th>
<th>Q13</th>
<th>Q14</th>
<th>Q15</th>
<th>Q16</th>
<th>Q17</th>
<th>Q18</th>
<th>Q19</th>
<th>Q20</th>
<th>Q21</th>
<th>Q22</th>
</tr>
</thead>
<tbody>
<tr>
<td>PartitionApproach(20)</td>
<td>211</td>
<td>223</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
<td>5,481</td>
<td>&gt;10mins</td>
<td>5,198</td>
<td>&gt;10mins</td>
<td>126,180</td>
<td>151,728</td>
</tr>
<tr>
<td>PartitionApproach(40)</td>
<td>263</td>
<td>307</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
<td>61,438</td>
<td>&gt;10mins</td>
<td>9,189</td>
<td>&gt;10mins</td>
<td>197,396</td>
<td>140,133</td>
</tr>
<tr>
<td>PartitionApproach(60)</td>
<td>263</td>
<td>1,452</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
<td>52,423</td>
<td>&gt;10mins</td>
<td>10,492</td>
<td>&gt;10mins</td>
<td>124,019</td>
<td>132,178</td>
</tr>
<tr>
<td>PartitionApproach(80)</td>
<td>267</td>
<td>1,200</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
<td>78,719</td>
<td>&gt;10mins</td>
<td>10,215</td>
<td>&gt;10mins</td>
<td>114,020</td>
<td>161,539</td>
</tr>
<tr>
<td>PartitionApproach(100)</td>
<td>267</td>
<td>1,134</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
<td>53,596</td>
<td>&gt;10mins</td>
<td>290,967</td>
<td>&gt;10mins</td>
<td>112,665</td>
<td>166,413</td>
</tr>
<tr>
<td>NodeApproach</td>
<td>118</td>
<td>259</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
<td>&gt;10mins</td>
</tr>
<tr>
<td>BranchClassIndex</td>
<td>16</td>
<td>92</td>
<td>63</td>
<td>81</td>
<td>3,274</td>
<td>896</td>
<td>229</td>
<td>1,371</td>
<td>192</td>
<td>996</td>
</tr>
</tbody>
</table>

Table 7.15: Results for the Node Based Approaches

In Table 7.15, the query response time for each of these queries is shown. These results show the following:

- The NodeApproach timed out on all but Q13, Q14, Q18.
- In the PartitionApproach, a partitioning factor of 100 returned re-
results for the greatest number of queries (Q13, Q14, Q16, Q17, Q19, Q20, Q21, Q22). Query Q19 shows an increase in processing times as the partitioning factor increased, whereas Q21 shows a decrease. The remaining results do not suggest such a pattern.

- The PartitionApproach returned results for a greater number of queries then the NodeApproach across all partitioning factors.
- The BranchClassIndex is orders of magnitude faster across all queries.

Queries Q13 and Q14 have high selectivity as they return a single result node. Also, the first two steps in Q19, that is, /site and /closed_auctions, both evaluate just a single node only. We attribute the fact that NodeApproach returned results for queries Q13, Q14 and Q15 to the high selectivity of these queries.

There is no consistent pattern between the incrementing partitioning factors indicating that no single partitioning factor per dataset is ideal. The PartitionApproach provides superior results than the NodeApproach, both in terms of query response times, and in terms of the number of queries that returned a result within 10 minutes. However, the exhaustive experimentation required to identify suitable partition factors is infeasible (in terms of index build times and query analysis). Both approaches do not scale well for queries that have low selectivity, because even for relatively small XML datasets (115 MB), the query response times are large relative to those of the BranchClassIndex.

### 7.6 Summary

In this chapter, we evaluated the performance of the BranchClassIndex; a path-based indexing approach (PathBasedIndex); a leading open-source (MonetDB/XQuery); and a commercial (SQL Server 2008) XML database.
Linear path queries (QC\textsubscript{1}) do not contain predicate filters, therefore all approaches are effective in this category. Across all datasets, SQL Server performed marginally better than the BranchClassIndex, followed by the PathBasedIndex and finally MonetDB/XQuery is the poorest performing approach in this category.

The BranchClassIndex is the most effective approach for queries in QC\textsubscript{2} for all but the XMark dataset. For queries executed across the XMark dataset, MonetDB/XQuery is more efficient. The reason for this is the increased number of ancestor/descendant and parent/child relationships. We suggest ways of minimising this increase to the index size in our discussion on future research (Chapter 8).

For Twig queries that evaluate text values with low selectivity QC\textsubscript{3}, the BranchClassIndex performs best for the protein sequence database, whereas MonetDB/XQuery performs best for the bicycle rental and computer science bibliography datasets (the XPathMark evaluation did not contain queries in this category). However, for the bicycle rental dataset, the BranchClassIndex is 162 times faster than MonetDB/XQuery when a small subset of text values are indexed and 12 times faster when a large subset of text values are indexed. For the computer science bibliography, the BranchClassIndex is 83 times faster than MonetDB/XQuery after all low selectivity text values have been indexed.

For queries that evaluate text values that have high selectivity (QC\textsubscript{4}), SQL Server performed best for the bicycle rental and XMark datasets, followed by the BranchClassIndex, MonetDB/XQuery and the PathBasedIndex respectively. For the computer science bibliography and protein sequence dataset, the BranchClassIndex is the most efficient approach.

Overall for the bicycle rental dataset and protein sequence datasets, MonetDB/XQuery is the most efficient approach. However, for the bicycle rental dataset, the BranchClassIndex replaces MonetDB/XQuery as the most ef-
ficient approach when a small or a large subset of text values are indexed. The overall averages for the XMark evaluation show that the BranchClassIndex and MonetDB/XQuery perform at the same level, whereas the PathBasedIndex and SQL Server are less efficient by orders of magnitude. For the computer science bibliography, the BranchClassIndex performs best overall when all low selectivity text nodes are indexed.

These experiments show that the BranchClassIndex is a poor choice for one query category (QC$_3$). This is because queries in this category contain text values that have low selectivity, thus leading to large numbers of node comparisons. However, it was shown that indexing low selectivity text values improves the performance of the BranchClassIndex making it the most efficient approach overall for the bicycle rental and computer science bibliography datasets. For datasets where the BranchClassIndex did not perform best overall (XMark and protein sequence datasets), its storage costs are higher. Therefore, as part of our future work, we discuss ways in which the BranchClassIndex can be compacted.
Chapter 8

Conclusions

This dissertation presented a node partitioning strategy for optimising XPath queries. In this chapter, we restate the concepts that were introduced and discuss future work. This chapter is structured as follows: in §8.1, a summary of the dissertation is provided and in §8.2, the future potential of this work is discussed.

8.1 Thesis Summary

The partitioning approach, which to the best of our knowledge is shared with just one other index-based approach [43], allows nodes of different names and types (element and attribute) to reside in the same partition. The hypothesis is based on the fact that there will always be fewer partitions then there are nodes. Therefore, the partitions that contain target nodes can be identified directly and all other partitions are eliminated from the search space. In contrast to [43], however, our approach allows partitions of different sizes within an XML document and avoids time-consuming preprocessing to identify suitable partitioning factors.

The partitioning approach optimises the most commonly used XPath axes (the six hierarchical axes). We began by defining a disjoint partition of nodes
as a *branch* because suitable partitions are identified based on the layout of *branching nodes* (nodes that have at least two children) within the XML document. It soon became clear that the most suitable partitioning strategy for the hierarchical axes was to ensure that only nodes that are hierarchically related can reside in the same branch. We showed that this has the effect of reducing the number of false hits, thus improving the performance of the hierarchical axes.

A side effect of using the rule that only hierarchically related nodes can share the same branch is that there may be a large number of branch instances. Thus, because the optimisation strategy is based on the fact that there will be fewer partitions than nodes, the increased number of branch instances may result in a performance overhead. To overcome this issue, we provided a classification process for equivalent branch instances. After classification, a single branch class represents any number of branch instances. In effect, the index is compacted while maintaining its search space pruning benefits. Based on the fact that a single branch class represents many branch instances, we then showed that a single branch class proxy can be processed in place of large numbers of branch instances for many steps in XPath expressions. As a result, we showed that for many XPath expressions the majority of nodes are bypassed during the query process.

After the branch classification process, we described how the index can be deployed in a relational database. We provided a detailed description of the relations that were used in its deployment and detailed an XPath-to-SQL transformation process for the core subset of the XPath language.

Finally, we demonstrated the performance of the index using four large XML datasets. A real world dataset was used to demonstrate the wider impact of our approach. Then, we exploited three datasets that are commonly used by the research community for benchmarking purposes.
8.2 Future Work

In this section, we describe the long and short term goal of our on-going research.

8.2.1 Reducing Redundancy

In this dissertation, we showed how the ClassChain components can be stored in relations that are suitable for equijoin evaluations; these relations
are shown in Figure 8.1b3 and 8.1b4. We chose this approach because the author of [41] showed us that relational databases can evaluate equijoins efficiently (unlike non-equijoins).

An issue associated with a relational storage method that is suitable for equijoin evaluation is redundancy (see Chapter 2 for more details). The approach presented in this dissertation reduces redundancy to a large degree through branch classification, but we believe it can be reduced further.

Consider node 1 in Figure 8.1a (city). This node’s branch class identifier (C11) is an ancestor class of all other branch classes in the XML document (except the class that the document node belongs to). Redundancy occurs because C11’s descendant classes (highlighted in grey in Figure 8.1b3 and 8.1b4) also occur as descendants of other branch classes. For example, class C9 is a descendant of class C11 and C10.

![Figure 8.2: Redundancy Reduction](image)

Only branch classes that occur at low levels in the XML document that cause significant redundancy issues. For example, the class for node city (level one) in the bicycle rental dataset or regions (level one) in the XMark
dataset. We believe that it is possible to identify such branch classes within the XML document and reduce the impact on redundancy for the following reason.

A branch class such as \(C_{11}\) is an ancestor of classes \(C_1-C_{11}\). Another way of looking at it is that branch class \(C_{11}\) is a parent of classes \(C_5\) and \(C_{10}\), and is an ancestor of \(C_5\) and \(C_{10}\)’s descendants. Thus, we propose the new relational layout shown in Figure 8.2. The special case branch classes that are ancestors of a large number of branch classes can be placed in a separate relation as shown for class \(C_{11}\) in Figure 8.2b1 and the redundant duplications can be removed from 8.2b2 and 8.2b3 as shown. We also propose that this is the approach that will be used to include the document node in future versions of the BranchClassIndex.

The XPath-to-SQL transformation process would need to be updated to perform an additional check across such a new relation. However, we don’t foresee this to be a difficult problem, and we project that the number of special case nodes would be small within each XML document, thus the additional check should not be too costly.

### 8.2.2 Other Future Directions

In this dissertation, we have described an optimisation strategy for the hierarchical XPath axes. We focused on the hierarchical axes because, in our experience, they are the most commonly used axes. However, integration of other axes such as \(attribute\) should require only simple extensions to the XPath-to-SQL transformation process. In fact, the attribute axis should already benefit from our approach as attribute nodes are placed in the same branch class as their parent element. Thus, identifying an element node’s attributes will be the efficient task of identifying the attribute nodes in the same branch class as itself.

In contrast, to date we have not seen any real world requirements for the
*following* and *preceding* XPath axes. Thus, we propose addressing these axes in the future if we can motivate the problem based on further research. Also, we intend to determine the benefit of our branch partitioning approach for the broader XQuery language.

Finally, our ultimate goal is determine and overcome the overhead of maintaining the BranchClassIndex upon XML updates. The BranchClassIndex is encoding scheme independent, thus it can be used along with more update friendly encoding schemes such as [10,55]. However, the introduction of new nodes may invalidate branch classes. Alternatively, it could be that new nodes are inserted into already existing branch classes leading to little index maintenance overhead.
Bibliography


