Child Prime Label Approaches to Evaluate XML Structured Queries

Shtwai Abdullah Alsubai

Department of Computer Science
the University of Sheffield

This thesis is submitted for the degree of
Doctor of Philosophy

March 2018

I would like to dedicate this thesis to my beloved father, mother, wife and son.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This thesis is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text. This thesis contains fewer than 130,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 Figures.

Shtwai Abdullah Alsubai
March 2018

Acknowledgements

First and foremost, all praises to ALLAH for the strengths and blessing in completing this research.

At the completion of this work I would like to acknowledge my supervisor Dr. Siobhán North for her guidance and continuous support in overcoming numerous obstacles I have been facing through my research. I am also grateful to the members of my committee for their valuable guidance during our meetings the last three years.

I would like to express my very profound gratitude to my parents and wife for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. Their unconditional love, patience and consistent support enable me to complete this research. A special thank is due to the joy of my life, my son Abdullah who has given me much happiness and keeps me hopping.

Last but not least, many thanks for brothers and sisters for supporting me spiritually throughout writing this thesis and my life in general.

Abstract

The adoption of the eXtensible Markup Language (XML) as the standard format to store and exchange semi-structure data has been gaining momentum. The growing number of XML documents leads to the need for appropriate XML querying algorithms which are able to retrieve XML data efficiently. Due to the importance of *twig pattern matching* in XML retrieval systems, finding all matching occurrences of a tree pattern query in an XML document is often considered as a specific task for XML databases as well as a core operation in XML query processing.

This thesis presents a design and implementation of a new indexing technique, called the Child Prime Label (CPL) which exploits the property of prime numbers to identify Parent-Child (P-C) edges in twig pattern queries (TPQs) during query evaluation. The CPL approach can be incorporated efficiently within the existing labelling schemes. The major contributions of this thesis can be seen as a set of novel twig matching algorithms which apply the CPL approach and focus on reducing the overhead of storing useless elements and performing unnecessary computations during the output enumeration. The research presented here is the first to provide an efficient and general solution for TPQs containing ordering constraints and positional predicates specified by the XML query languages.

To evaluate the CPL approaches, the *holistic* model was implemented as an experimental prototype in which the approaches proposed are compared against state-of-the-art holistic twig algorithms. Extensive performance studies on various real-world and artificial datasets were conducted to demonstrate the significant improvement of the CPL approaches over the previous indexing and querying methods. The experimental results demonstrate the validity and improvements of the new algorithms over other related methods on common various subclasses of TPQs. Moreover, the scalability tests reveal that the new algorithms are more suitable for processing large XML datasets.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Introduction

The eXtensible Xarkup Language (XML) models semi-structured data and has emerged as the main standard for the representation and exchange of data over the internet. There are two basic strategies currently being adopted in research into storing XML data [154, 24]. One is the *XML-enabled* database which uses well established models to store and query XML data (e.g., relational and object-oriented database systems) and the other is the native XML database which is mainly designed to manage the organization, storage, access and security of XML data. This study attempts to improve XML querying and retrieval in native XML databases.

The growing number of XML documents leads to the need for appropriate XML querying algorithms which are able to utilize the specific characteristics of XML documents. Twig pattern matching (TPM) is a major area of interest within the field of XML query processing. Thus, TPM is a core operation in XML query processing because it is how all the matching occurrences of a twig pattern (i.e., a labelled query tree) in an XML tree (i,e., an XML document is modelled as a rooted-labelled tree) are found. In the last decade, several querying algorithms have been proposed to perform twig pattern matching [40, 144, 185, 22, 132, 89, 11, 8, 30]. However, it could be argued that most of the existing algorithms fail to process XML twig pattern queries efficiently in terms of processing time capabilities and space overheads. Therefore, the work of this thesis investigates the factors that may improve the efficiency of the TPM approach regarding the worst-case I/O complexity.

The research begins by laying out the basics of XML, and investigates up-to-date approaches for querying XML data in native XML databases leading to the identification of limitations in the existing methods which in turn motivate further work. This study set out to propose new XML query processing approaches to overcome these drawbacks. To reduce the memory consumption and computation overhead of twig pattern matching algorithms, the approaches proposed are based on combinations of a novel indexing

technique which exploits the property of prime numbers and the existing twig matching algorithms.

The purpose of this short chapter is to give an overview of the thesis. Section 1.2 outlines the thesis structure. a list of published work and research activities is provided in Section 1.3. The chapter is then concluded in Section 1.4.

1.2 Thesis Outline

The rest of this thesis is structured as follows: Chapter 2 aims to introduce the concept of XML and its related concepts. Most of the related models and techniques used in this thesis will be briefly discussed.

Chapter 3 provides a review of literature related to twig pattern query (TPQ) processing. The chapter begins by reviewing the XML processing techniques as a prerequisite to demonstrate XML query processing model including XML Indexing and partitioning. It will then go on to provide an overview of previous research on twig pattern matching including approaches proposed and their complexities.

In Chapter 4, problems arising from existing approaches are discussed in order to identify a place where a new contribution could be made. Consequently, the research hypothesis is highlighted. After that, this study justifies the choice of research method in order to test the hypothesis. Furthermore, the scope of the research and the expected objectives are discussed.

Chapter 5 introduces the design of experiments which are described in this thesis in order to compare the performance of the algorithms proposed to the related methods in the literature. It also describes the objectives of the experiments and the statistical procedures used to analyse data collected in experiments.

Chapter 6 deals with introducing a new approach to index Parent-Child edges in XML trees. After that, a novel holistic twig matching algorithm which is based on this technique is proposed and compared to the existing algorithms in the literature. This chapter consists of two parts: the first part shows the main drawbacks in the existing approaches and presents the new indexing and querying methods. The second part aims to discuss the experiments and the overall evaluation in order to draw a conclusion using statistical procedures. Some of the contents of this chapter were published in previous papers [12, 11].

Chapter 7 describes the process of ordered twig patterns in which normal twig patterns contain order and sequence constraints. Little research work has dealt with this type of twig patterns and most of them assumed twig patterns are ordered as a whole tree without considering the semantics of order axes introduced in the XPath specification. This chapter examines the emerging role of the semantics of order axes in the context of TPQ matching. The algorithms proposed are the first to consider sequence operators when processing holistic twig matching algorithms. The performance of the algorithms proposed

are compared to the existing approaches in the literature which are modified in order to process ordered twig patterns and sequence operators.

Chapter 8 describes how the advanced preorder filtering strategy proposed in this study is applied to bottom-up twig matching algorithms in order to improve query performance. A set of twig matching algorithms are proposed in order to enhance the filtering phase and speed up the query response time. A set of experiments are described to compare the performance of the new approaches against other related bottom-up holistic twig matching methods. The experimental results are analysed and discussed to evaluate the research hypothesis.

In Chapter 9, bottom-up holistic twig matching algorithms for processing ordered twig patterns are proposed. The experiment is designed to compare the performance of different approaches and demonstrate the improvements of the algorithms proposed. This chapter also discusses the new approach to process twig patterns with positional predicates. A novel method is introduced and compared to the related algorithms in the literature. Lastly, the experimental results are analysed and conclusions are drawn at the end.

In Chapter 10, the thesis presents and evaluates the designs and results of the four experiments conducted from different perspectives. In addition, it discusses the features and limitations of the experiments. The chapter is concluded by identifying the main findings of the experiments.

Chapter 11 is the final chapter in this thesis and aims to conclude the research work presented in this thesis by summarising it and identifying possible directions for future work.

1.3 Publications

What follows is a list of publications and research activities related to this research study.

1.3.1 Poster

1. S. Alsubai and S. North,“TwigList+: A New Approach for Processing XML Twig Queries” in Computer Science department, The University of Sheffield, UK, during the research retreat event in June 2015.

1.3.2 Peer Reviewed Papers

2. S. Alsubai and S. North, “A Prime Number Approach to Matching an XML Twig Pattern including Parent-Child Edges”, in The 13th International Conference on Web Information Systems and Technologies (WEBIST 2017), 2017, pp. 204–211.

3. S. Alsubai and S. North, “TwigStackPrime : A Novel Twig Join Algorithm Based on Prime Numbers”. In Lecture Note Business Information Processing, Revised Selected Papers WEBIST, Springer, 2018, 20 pages (in press).

1.4 Conclusion

This research aims to improve the performance of the XML query processing algorithms in native XML databases. After surveying the related work of XML query processing and optimisation in Chapter 3 and discussing the problem identifications, the research objectives and contributions will be discussed later in Chapters 4 and 11, respectively.

This chapter presented introductory information about the thesis and outlined the thesis structure by giving a brief summary of each chapter. The next chapter introduces the general background of XML and explains its related concepts including the semantics of XML documents, data model, XML parsers and XML query languages.

Chapter 2

XML Databases' Background

2.1 Introduction

Extensible Mark-up Language (XML) is emerging as a de facto standard for information exchange among various application on the World Wide Web due to its capability in organizing data and because of it is self-describing [81, 225, 98, 239]. As the volume and the number of XML documents get larger because of the universal adoption of it over the internet and digital libraries, the term XML database is a relatively new name for semi-structured databases [107]. An XML database can be defined as a database that is comprised of a well-formed XML document or a set of such documents [180, 45, 86, 209]. XML databases may be divided into *native* and *enabled* databases depending on whether the support of the XML data is the initial feature of the system or it is added later [143]. XML databases are still an emerging concept that has not been considered yet as a well-established field in comparison to traditional Relational Databases (RDBs) or Object-Oriented Databases (OODBs). It is a widely held view [98, 154, 107, 72, 150] that new strategies and techniques to support storing and querying XML databases is a fundamental need. Therefore, the process of retrieving data within the XML database has attracted a considerable number of researchers over the past decade [5, 40, 188, 98, 68].

This introductory section provides a brief overview of XML databases, it then goes on to describe the concepts of XML databases in Section 2.2. In Section 2.3 and Section 2.4, the processes of parsing and querying XML data will be explained in greater detail. The chapter will be concluded in Section 2.5.

2.2 The Concepts of XML Databases

The Extensible Markup Language (XML) models semi-structured data, and is the standard for sharing, saving and exchanging business data over heterogeneous and homogeneous platforms [81, 225, 98, 239]. There are two main approaches to storing XML documents. On the one hand, XML documents can be stored using one of the conventional databases such as relational or object-oriented as the underlying storage mechanism. This approach

is called XML enabled-databases (as abbreviated XEDs). On the other hand, storing XML documents in its own format without mapping the data to a different model is known as the native storage technique. XML databases that store XML documents in the native format are called native XML databases (NXDs, for short) [98, 250, 154, 150, 204, 197].

This section aims to outline the underlying concepts of XML databases and provides a general background for XML data management. In the subsections that follow, it will be explained what is the difference between XML enabled databases in Section 2.2.1 and native XML databases in Section 2.2.2. Also, more explanation of some important terminologies in XML databases will be described in Sections 2.2.3, 2.2.4 and 2.2.5, namely: the syntax rules of XML, the tree representation of XML data and XML schema, respectively [225, 120, 226].

2.2.1 XML Enabled Databases (XED)

According to [110] three main fundamental functions have to be provided when working on XML data. These are: adding information to the repository, retrieving information from the repository and updating information in the repository. An outstanding database must perform them very well. XML enabled databases are powerful in terms of providing the database features such as scalability, portability, recovery control, query engine and so on [211, 87, 98, 154, 187]. Because XML enabled databases store XML documents in another format such as tables as in relational database or objects as in object database, most of the works in the literature have focused on using relational database as the underlying storage [3, 236, 235]. This may be due to the limitations that the object oriented data model does not have standard object-oriented query language nor does it have a research prototype of XML data storage as object oriented is the underlying storage [98, 187, 197]. However, there exists a mismatch between the XML semi-structured data and relational data, hence, mapping plays an important role in providing seamless integration between these database infrastructures. This process is referred to as the *shredding* of XML document into tables. It is essential for a mapping strategy to preserve the hierarchical structures of XML documents during the mapping process in order to process XML queries effectively [197, 126] so that structured query language (SQL) can be used to query XML data. Several different approaches for mapping XML documents into relational schemas have been discussed and proposed over the last few years [76, 211, 56, 98, 204, 177, 49, 191, 56, 187, 197]. In spite of XEDs benefits in terms of providing database features (such as availability, scalability, security, concurrency and so on), the key problem with these approaches is that XML queries can not be processed efficiently, particularly for a large XML document. However, further discussion on this issue is outside the scope of this thesis.

2.2.2 Native XML Databases (NXD)

A native XML database (NXD) is a customized database that is built from scratch to store and handle XML documents in their original format [98, 154, 73, 107, 86]. A straightforward strategy to store an XML document is to store it directly in its textual format, called text-based model (also known as document-based storage). The whole XML document is stored as a file either in the file system of a server or in LOB (Large Object) columns in the DBMS. One major drawback of this approach is that it can only handle XML data up to 2 GB when the XML file stored in LOB types, and inefficient query performance due to the fact that the entire file has to be loaded and scanned by the query processor in order to execute the query [27, 126, 197]. In contrast, another strategy stores the XML document into an internal data structure as the Document Object Model (DOM) or the events as defined by the Simple API for XML (SAX) [225, 194]. This technique is called model-based storage (also known as node-based). Since the representation of XML data in this model is mainly structure oriented which has a pivotal role in query formulation, native XML indexes [85, 40, 188, 5, 52, 95, 246, 24, 154, 102] have been introduced to index XML structures in order to improve the efficiency and scalability of query processing. Among this literature, much published research has been proposed to improve the performance of XML query processing. This research study falls in this area. The review of native XML technologies used to evaluate XML queries will be presented in Chapter 3.

Furthermore, [150] studied and analysed the different model approaches for storing a native XML database and evaluated the performance for each approach on a set of commercial native XML database products available in the market at that time. The empirical comparisons show that using a node-based model provides more flexibility and performance for the entire database system. In addition, [154] addressed the performance degradation for indexing and storing in native XML databases. The best examples for a native XML database approach are research prototypes of XML data storage such as in [144, 89, 22, 7, 40]. A major advantage of NXDs is that they can provide a more natural data model and query language for XML data, which is typically represented as a graph (i.e., tree-based structure).

2.2.3 XML Syntax

XML is Mark-up language and is gaining popularity for data representation and exchange. Nested, tagged elements are the building blocks of XML documents. Due to the definition of relationships in XML as nested tags, data in XML documents are self-describing and flexibly organized [132]. For example, consider the XML document[1] in Figure 2.1, the document is composed of two main components, namely: element and attribute. It contains a declaration that identifies the document as an XML document, called XML prolog in Line

[1]DBLP document snippet obtained from http://dblp.uni-trier.de/

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <dblp>
3  ...
4  <inproceedings key="conf/webist/AlsubaiN17">
5  <author>Shtwai Alsubai</author>
6  <author>Siobh&aacute;n North</author>
7  <!-- aacute is a character entity reference for latin small letter a
       with acute accent -->
8  <title>A Prime Number Approach to Matching an XML Twig Pattern
       including Parent-Child Edges.</title>
9  <pages>204-211</pages>
10 <year>2017</year>
11 <booktitle>WEBIST</booktitle>
12 <ee>https://doi.org/10.5220/0006225602040211</ee>
13 <crossref>conf/webist/2017</crossref>
14 <url>db/conf/webist/webist2017.html#AlsubaiN17</url>
15 </inproceedings>
16 ...
17 </dblp>
```

Figure 2.1: An example of XML data representing an inproceedings element in the DBLP document.

1. XML makes use of one built-in element starting with XML as in Line 1 to validate some special meanings beside informing the system to expect an XML document to accelerate the document processing (this will be explained more in Section 2.3). Unlike Hyper Text Mark-up Language (HTML), the syntax rules of XML are strict. As a result, an XML document is considered as a well-formed document only if it conforms to the syntax rules (constraints) and they are simple and flexible as follows:

- The document must have one element that identifies the entire document, called root element (see Section 2.2.3.1).
- For every opening tag, there must be a corresponding closing tag (see Section 2.2.3.1).
- The sequence of distinct attribute names within a single element must be unique (see Section 2.2.3.2).
- The value of every attribute must be quoted using either single or double quotation marks (see Section 2.2.3.2).

An XML documents is comprised of five components including elements, comments, attributes, namespaces and processing instructions. The following subsections provide a description of XML elements and attributes which are the main two components of XML files [225, 120, 226].

2.2.3.1 XML Elements

An element is deemed as the basic component of information in XML documents. Consider the example in Figure 2.1, it starts with an opening tag as in Line 2 $< dblp >$ and ends

with a closing tag as in Line 17 $< /dblp >$, this is the root element. Each XML document has a single root element. An element may contain atomic data as in Line 5 for the element $< author >$, other elements as in Line 4 the element $< inproceedings >$ contains a set of elements, or both (i.e., mixed content). Also, an element might be empty with nothing enclosed between the pair of tags and in this case the element can be defined using one tag as $< tag/ >$. An element can have an attribute that is included within the opening tag to describe the specific element precisely as in Line 4 the element $< inproceedings >$ is assigned with *key* as an attribute. Furthermore, as with other languages, XML supports comments to contain more information and provide human-readable annotations about the actual content. There are two kinds of information which can be found in XML files: markup (e.g., <dblp>) and the character data (e.g., 2017). In XML comments are always considered as child elements and start with a specific opening tag $\{<!--\}$ and end with a specific closing tag $\{-->\}$, see Line 7 of Figure 2.1 [203, 226, 120].

2.2.3.2 XML Attributes

Generally, attributes are used to provide additional information about elements, especially when the information is irrelevant to data contained within elements. For example, the *key* attribute in Figure 2.1 is for identifying the different *inproceedings* in the DBLP document. Attributes are added to elements, so they are dependent on their elements. Attributes only can have value and that value is declared within either single or double quotation marks. Attributes are placed within the opening tags of elements and an element may have a sequence of zero or more attributes. This can be demonstrated in Line 4 of Figure 2.1 as $< inproceedings\ key = "conf/webist/AlsubaiN17" >$. There are a number of important differences between elements and attributes. While elements can contain multiple values and tree structure, attributes can not. However, an element can only have a single attribute of type *id* whose value provides a unique identifier that can be referenced by attributes of type id_{ref} from other elements. Attributes are widely-used as parameters to sort XML documents because they are designed to contain data related to a specific element [203, 226, 120]. For example, the DBLP document can be sorted by type of records (books, journal articles, conference and workshop papers, etc.) using the *key* attribute that further specifies the type of record. An XML namespace can be loosely described as a collection of names that is identified by a URI (Uniform Resource Identifier) reference to avoid tag name conflicts in XML by using prefix. The namespace can be defined by a predefined attribute, *xmlns* in the start tag of an element [226].

2.2.4 XML Tree Structure

The XML is a textual language rather than a data model [86]. Because of that an XML document has an implicit order. The internal structure of an XML document might be very complex and get more complicated since that order cannot be avoided. Therefore, XML is

commonly modelled as tree structure for more simplicity in the document representation as $T = (N, E, r)$, where N is a set of nodes, $r \in N$ is the root of T, and E is a set of edges connecting couples of nodes together $(n_i, n_j) \in N \times N$. Tree nodes are labelled by elements, attributes or atomic data (also known as leaf nodes). Tree edges represent element-element, element-attribute and element-value relationships (also known as Parent-Child relationships, P-C for short). Figure 2.2 shows the tree-based representation for the XML document in Figure 2.1. It can be seen that the need for opening and closing tags in XML is unavoidable and might increase the complexity of the document's internal structure because the nesting of elements renders the overall tree structure of XML documents.

Although an XML tree provides easy document access, a large amount of memory is consumed by the document when the document is modelled as tree. More relationships can be defined between the elements within the XML document such as Ancestor-Descendant (abbreviated as A-D) and sibling (this can be more divided into two more specific relationships: *preceding-sibling* and *following-sibling*) relationships. An Ancestor-Descendant relationship is structural information in a tree that indicates one element is contained within another element through one or more elements. For example, in Figure 2.2 *pages* is a descendant of *dblp*. A sibling relationship is defined as a relationship between two or more elements that have the same parent. Consider Figure 2.2 as an example, the elements *author* and *booktitle* share the same parent, *inproceedings*.

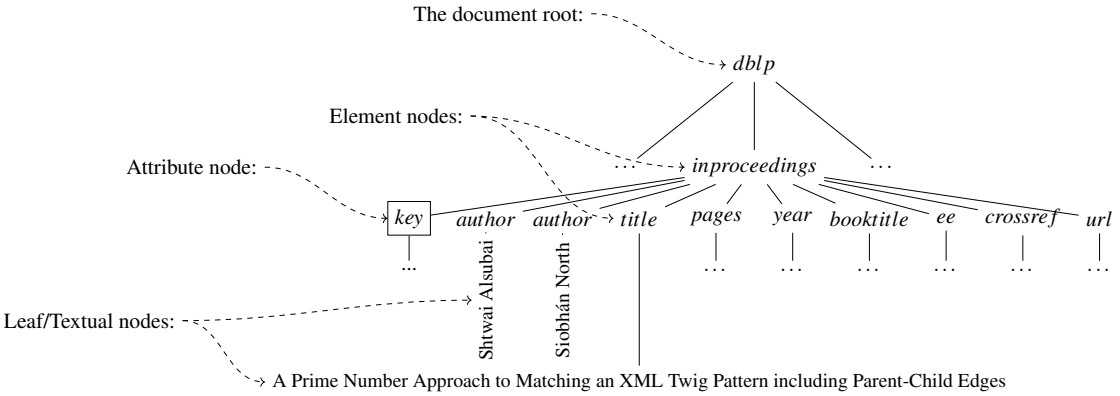


Figure 2.2: The tree representation of the XML document in Figure 2.1

It is necessary here to clarify exactly what is meant by *document order* which is defined among all the tree nodes accessible during XML processing. Generally, *document order* is the order in which XML elements and attributes are accessed during XML processing and the output generation. *Document order* must satisfy the following constraints within an XML tree.

- The document root is the first node.
- Every parent node must precede its children and its descendants in the document order.
- Namespace nodes must be ordered immediately after their parents.

- Attributes must be ordered after their parents and their sibling namespace nodes (if they exist).
- Node's children and descendants are ordered before its following siblings.
- The children property of a parent node determines the relative order of its children (i.e., the relative order of siblings).

2.2.5 XML Schema

A schema can be loosely defined as a way to describe an organised pattern of thoughts or actions that constitute collections of information and the relationships among them [203].

In the context of XML databases, the purpose of an XML schema is to define legal building blocks of an XML document and provide a set of built-in data-types [223, 203, 71, 1]. There exist several XML schema languages for expressing constraints about XML documents, yet the main two languages that are supported in widespread use and have high degree of interoperability, Document Type Definition (DTD) and XML Schema Definition (XSD). The last one is recommended by W3C [223, 167, 79, 3, 72, 228, 1, 152]. Figure 2.3 shows an example of an XML schema document of DTD and XSD describing the XML document in Figure 2.1, respectively. The key features of an XML schema can be listed as follows:

- to provide a list of the main components in an XML document: elements and attributes.
- to constrain where elements and attributes can occur within an XML document.
- to show what data-types such as String and Integer must be assigned to leaf elements.

2.2.6 Summary

With regard to XML databases, underlying storage techniques have a significant impact on their performance [150, 154, 204, 3]. In this section an overview over the different storage techniques have been discussed. The efficiency of the underlying storage is measured by different criterion: the ability of the storage mechanism to retrieve data effectively and accurately and exploit the storage resources efficiently [98]. As has been mentioned above in Section 2.2.1 and Section 2.2.2, many approaches have been proposed in the literature to improve the process of storing XML documents, hence, enhancing the process of retrieving information from a large repository of XML documents (this will be explained more in Section 2.4).

Listing 2.1: DTD

```
<!ELEMENT dblp (article|inproceedings|proceedings|book|incollection|
phdthesis|mastersthesis|www|person|data)*>
<!ATTLIST dblp mdate CDATA #IMPLIED >

<!ENTITY % field "author|editor|title|booktitle|pages|year|address|journal|volume|number|month|url|ee|cdrom|cite
      |publisher|note|crossref|isbn|series|school|chapter|publnr">

<!ELEMENT article       (%field;)*>
<!ATTLIST article
key CDATA #REQUIRED
mdate CDATA #IMPLIED
publtype CDATA #IMPLIED
reviewid CDATA #IMPLIED
rating CDATA #IMPLIED
cdate CDATA #IMPLIED
>

<!ELEMENT inproceedings (%field;)*>
<!ATTLIST inproceedings key CDATA #REQUIRED
mdate CDATA #IMPLIED
publtype CDATA #IMPLIED
cdate CDATA #IMPLIED
>
```

Listing 2.2: XSD

```
<xsd:complexType name="dblp">
<xsd:choice maxOccurs="unbounded">
<xsd:element name="article" type="article" maxOccurs="unbounded"/>
<xsd:element name="book" type="book" maxOccurs="unbounded"/>
<xsd:element name="incollection" type="incollection" maxOccurs="unbounded"/>
<xsd:element name="inproceedings" type="inproceedings" maxOccurs="unbounded"/>
<xsd:element name="mastersthesis" type="mastersthesis" maxOccurs="unbounded"/>
<xsd:element name="phdthesis" type="phdthesis" maxOccurs="unbounded"/>
<xsd:element name="proceedings" type="proceedings" maxOccurs="unbounded"/>
<xsd:element name="www" type="www" maxOccurs="unbounded"/>
</xsd:choice>
</xsd:complexType>

<xsd:element name="dblp" type="dblp"/>

...
<xsd:complexType name="inproceedings">
<xsd:choice maxOccurs="unbounded">
<xsd:element name="author" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="booktitle" type="xsd:string"/>
<xsd:element name="cdrom" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="cite" type="cite" minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="crossref" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="ee" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="month" type="xsd:string" minOccurs="0"/>
<xsd:element name="note" type="xsd:string" minOccurs="0"/>
<xsd:element name="number" type="xsd:string" minOccurs="0"/>
<xsd:element name="pages" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="title" type="title"/>
<xsd:element name="url" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="year" type="xsd:string"/>
</xsd:choice>
<xsd:attribute name="key" type="xsd:string" use="required"/>
</xsd:complexType>
...
```

Figure 2.3: A fragment of the DBLP schema corresponding to data provided in Figure 2.1

2.3 XML Parsing

XML parsing is a process by which the input XML documents are scanned, broken down into smaller elements, and eventually built into a corresponding data representation [225, 194, 117, 121, 231, 232, 36].

Based on the inner data representation, there are two categories of XML parsing: tree-based APIs and event-based APIs. On the one hand, tree-based parsers map the XML documents into an inner tree data structure and allow the applications to navigate them in memory only when the parsing process is completed, a well-known example of this category is Document Object Model (DOM) recommended by World Wide Web Consortium [225]. On the other hand, event-based parsers report parsing events such as the start and end of elements (i.e., opening and closing tags, respectively) directly to

the application through callbacks, and do not usually build an internal tree, a notable example of it is SAX (SAX Project Organization) [36, 194, 97]. For example, consider a fragment of the XML document shown in Figure 2.1, Figure 2.4 and Figure 2.5 illustrate that DOM and SAX have distinct functions in representing an XML document. It provides an illustration for how the two groups mentioned above handle an XML document.

Listing 2.3: An XML snippet

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <inproceedings key="conf/webist/AlsubaiN17">
3  <author>Shtwai Alsubai</author>
4  <author>Siobh&aacute;n North</author>
5  </inproceedings>
```

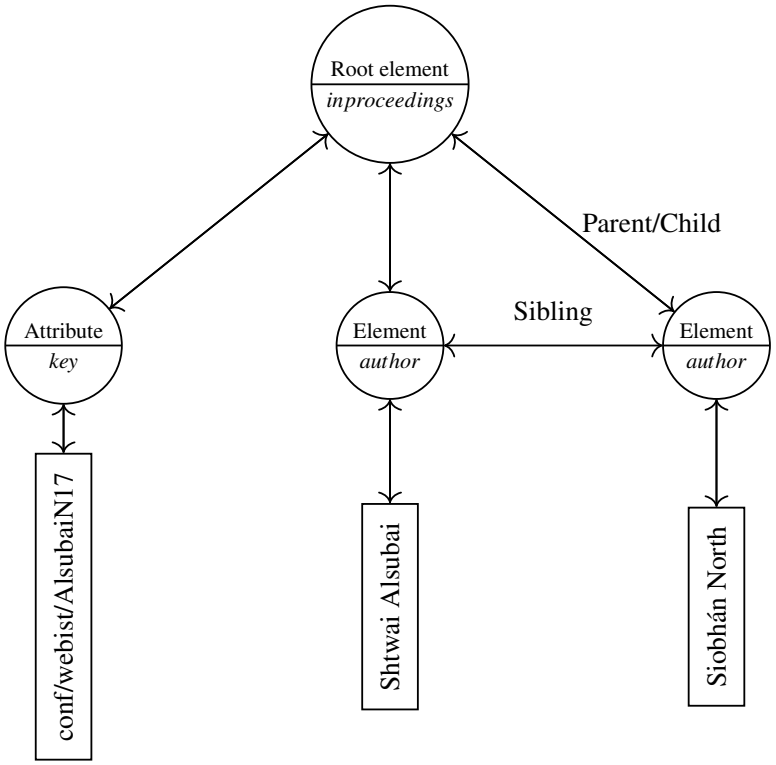


Figure 2.4: An illustration of the DOM tree for an XML file.

It can be seen from Figures 2.4 and 2.5, a DOM parser has to read through the entire XML document tree before an application can work on the output tree. One reason why DOM parser has difficulty parsing medium to large sized an XML document is that the entire document has to be presented in the main memory. Unlike a DOM parser, a SAX parser does not need to read through the entire XML document tree before allowing an application to process the XML data. It allows applications to construct customisable data structures using a series of predefined events [194]. As was mentioned earlier, a DOM parser may limit the size of XML documents processed, a SAX parser, on the other hand, can parse XML document trees of any size (e.g., number of XML nodes or disk-size) even if they are larger than the available resources [169].

XML parsing is a prerequisite for any processing of an XML document. As a consequence, XML parsing is both memory and computationally intensive, it consumes about

Listing 2.4: An XML snippet

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<inproceedings key="conf/webist/AlsubaiN17">
<author>Shtwai Alsubai</author>
<author>Siobh&aacute;n North</author>
</inproceedings>
```

Listing 2.5: SAX events

```
start document:
start element: inproceedings
attributes[1].name: key
attributes[1].value: conf/webist/AlsubaiN17
start element: author
characters: Shtwai Alsubai
end element: author
start element: author
characters: Siobhán North
end element: author
end element: inproceedings
end document:
```

Figure 2.5: SAX events for an XML document.

30 percent of processing time in many Web service applications. For instance, Morgan Stanley's Financial Service system spends 40 percent of its execution time on processing XML documents [16, 117]. The XML parsing process is considered as the bottleneck to performance when processing XML data. Many studies have been made to address the parsing performance through the use of pre-fetching, element-skipping and parallel approaches as in [117, 13, 231, 190].

2.4 XML Query

In general, a query is a question. In the field of computing, a query is a request for information from an organised collection of information (i.e, a database). In XML documents, information is organised by tag-names semantically and hierarchically. Using the tree structure of XML documents for information retrieval demands efficient support for a special query language to extract data from XML files. Therefore, several XML query languages have been designed such as XPath [222], XSLT [220], XPointer [221], Quilt [46] and XQuery [224]. These query languages are based on the concept of *regular path expression* (RPE) to specify path in a semi-structured data. The most two common, XPath and XQuery, are World Wide Web Consortium Recommendations (W3C) and are the common query languages for XML databases [24]. The following subsections provide a brief report on these two languages.

2.4.1 XPath

XPath stands for XML Path, which is an XML query language. XPath exploits path expressions to traverse an XML tree to select nodes or a set of nodes (also known as node-set). The mode of processing used by XPath is similar to that used in a traditional computer file system. In addition to addressing a specific portions of an XML document, XPath is used for matching (examining whether node matches a given pattern or not). The result of a path expression is a sequence of matching nodes. XPath models an XML document as a tree and supports structural relationships between XML nodes (e.g Parent/Child, Ancestor/Descendant, Sibling and etc ...). Moreover, numerous comparison expressions are supported such as value, logical, node and conditional comparisons [222, 42, 3, 46].

Starting from the context node which is defined as the current node in the XML tree, XPath 2.0 [222, 171] specifies a sequence of nodes (known as node-set) to be returned according to a given path expression. The path expression is the most important part of XPath and it is comprised of a number of axis steps, each of which consists of axis, *NodeTest* and zero or more predicates. The axis determines the structural relationship while *NodeTest* selects nodes based on their tag names and types. The predicate specifies additional constraints for filtering. The result set of the axis step will become the context node for the next axis step. The path expression might optionally begin with "/" or "//" which represent an implicit root node. The XPath language defines thirteen axes to traverse through the XML tree from the context node. Therefore, with respect to the context node, the corresponding XML tree is partitioned into a number of overlapping and distinct subtrees according to the XPath's axis specified. Explanation of the XPath's thirteen axes is given in Figure 2.6. Some of the XPath's axes can be expressed in an abbreviated syntax while used in path expressions, the commonly used abbreviated symbols are shown in Figure 2.7.

Every XPath expression which uses only P-C and A-D axes can be conveniently represented as tree pattern query. A tree pattern is a tree where nodes are tag names (i.e., elements and attributes) and structural relationships between nodes are specified by P-C ("/") and A-D ("//") edges. If any step is associated with a predicate expression (i.e., [ ]), it gives rise to a subtree corresponding to that step (i.e., branching). The node corresponding to the last step outside a predicate is the output node. Nodes (i.e., steps) can be constrained by value-based conditions. For example, consider the following XPath expression $p_1 = dblp//inproceedings[title]/author$. Figure 2.8 shows the tree representation of p_1, where P-C and A-D are visualised by single and double lines, respectively. The output node (i.e., author) is underlined to distinguish it from the other nodes. Throughout this thesis, the term twig pattern query (TPQ) will be used to refer to XML queries since it is the more general form of RPE [24].

As an illustration, the following are examples of XPath expressions to identify specific node-sets within the XML dataset shown in Figure 2.9, single characters represent element tags and subscripts identify instances of elements with the same tag. The following XPath

AxisName	Result
ancestor	contains the ancestors of the context node.
ancestor-or-self	contains the context node and the ancestors of the context node.
attribute	contains the attributes of the context node.
child	contains the children of the context node.
descendant	contains the descendants of the context node.
descendant-or-self	contains the context node and the descendants of the context node.
following	contains all nodes that are after the context node in document order, excluding any descendants, attributes and namespaces.
following-sibling	contains all the following siblings of the context node, excluding attributes and namespaces.
preceding	contains all nodes that are before the context node in document order, excluding any descendants, attributes and namespaces.
preceding-sibling	contains all the preceding siblings of the context node, excluding attributes and namespaces.
parent	contains the parent of the context node, if it is not the root.
self	contains the context node.
namespace	contains the namespace nodes of the context node.

Figure 2.6: The thirteen axes supported by XPath.

Expressions	Description
/	P-C relationship between two steps in a path.
//	A-D relationship between two steps in a path.
.	the current step.
..	the parent of the current step.
[ ]	predicate on the current step.
$@tag-name$	selected attribute of the current step.
$Position()$	positional constraint on the current step.

Figure 2.7: The most common symbols and functions used in XPath expressions.

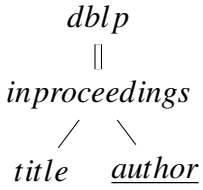


Figure 2.8: An example of XPath tree model

expression $p_2 = //e//a$ identifies all a-tagged nodes which are descendant of e-tagged nodes in the XML document. The result of this XPath expression is a set of ordered nodes $\{a_1, a_2, a_3, a_4, a_5\}$ because XPath traverses an XML document according to the document order which is the order the XML nodes would appear in a textual XML document. Tree model representation for p_2 is shown in Figure 2.10. On the other hand, the XPath expression $p_3 = //e/a$ finds all a-tagged nodes which are immediate child of e-tagged nodes. It returns the results $\{a_1, a_2, a_4, a_5\}$.

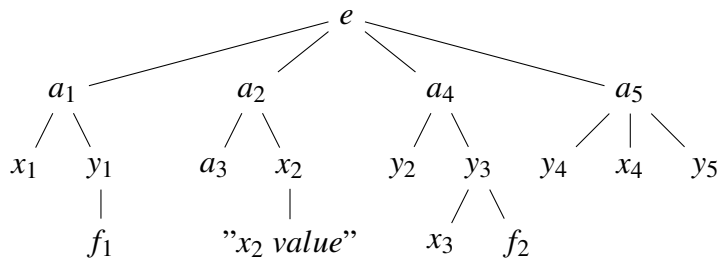


Figure 2.9: An example of XML tree T_1.

The next XPath expression $p_4 = //a[/y]/x$ finds all x-tagged nodes which are immediate child of a a-tagged node which has an immediate child y-tagged node. It is worth noting that some a-tagged nodes might not have siblings of x-tagged and y-tagged nodes. p_4 demonstrates the flexibility, in terms of structure, provided by semi-structured query languages such as XPath which is not available in the relational model. p_4 allows to search for subtrees rooted at a-tagged nodes without concern for where they are located within the XML data tree. Thus, the query results are $\{x_1, x_4\}$. The following XPath expression $p_5 = //e/a[/x = "x_2\ values"]$ returns a-tagged node which has an immediate child x-tagged node associated with content values as "x_2 *values*". The result of p_5 is $\{a_2\}$. In addition, predicates can be used to specify the numerical position of nodes within the sequence of nodes currently processed, these predicates are called positional predicates. The next XPath expression $p_6 = //e/a[//x]/y[position() = 2]$ refers to the second child y-tagged node of each a-tagged node which has x-tagged children in the XML tree T_1. The results of p_6 are $\{y_3, y_5\}$. Finally, XPath defines four ordered axes: following, following-sibling, preceding and preceding-sibling. To illustrate, the following-sibling axis can be used to select the following sibling x-tagged nodes after the y-tagged node in document order as in $p_7 = y/following-sibling :: x$. The evaluation of p_7 yields one result $\{x_4\}$.

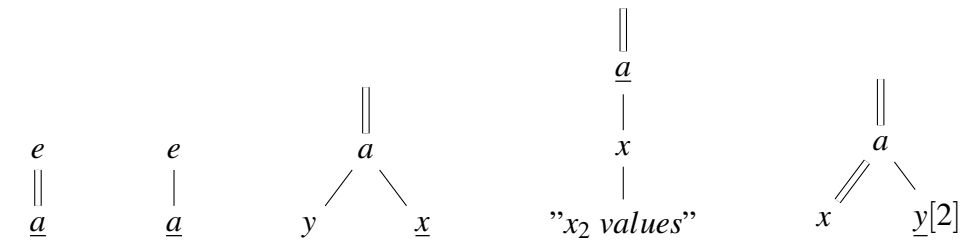


Figure 2.10: An example of XPath tree model

In XPath, all path expressions (i.e, XML queries) are evaluated against the tree representation of XML data. The XML queries represented in p_2, p_3 and p_5 are also called a single-path or a simple-path queries because it only consists of one leaf node as depicted in Figures 2.10a, 2.10b and 2.10d. However, the XPath expressions in p_4 and p_6 belong to

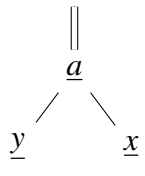
the set of queries (i.e., twigs) which define two or more leaves as shown in Figures. This will be discussed in more detail in Chapters 3 and 4.

2.4.2 XQuery

XQuery (XML Query Language) is an XML query language even though it can query structured and semi-structured data. It is specified by W3 Consortium XML Query Working Group and became a W3C Recommendation on January 23, 2007 [224]. XQuery is based on the existing XPath query language. Both XQuery and XPath share the same data model (tree-based structure) and support the same functionalities but XQuery is extended with features for better query expressiveness and to handle complex queries since XPath can only answer simple queries [224, 227, 75, 20]. It is worth noting that there is a slight difference between the semantics of XPath and XQuery. Whereas in XPath there is only a single query output node, XQuery can specify a set of query return nodes. However, this research focus on twig pattern matching which returns all possible combinations of node matches because it is practical for the flexibility in XQuery. Further, XQuery is considered as the most powerful XML query language for extracting information from an XML document or any collection of data modelled as XML. XQuery defines a set of expressions, called FLWOR expressions (For Let Where Order Return) to perform SQL-like (Structured Query Language) queries on XML data similar to that performed on relational databases. The *for* clause defines variables and iterates over nodes resulting from an XPath expression, while the *let* clause declares variables but does not iterate over them. The optional *where* clause specifies a selection condition similar to that in SQL. The optional *order by* clause can be used to specify the sorting order of the result. The *return* clause specifies what to be returned in the result. For instance, the following FLWOR expression shown in Figure 2.11a asks for the same data as the path expression p_4 presented in Section 2.4.1. In contrast to p_4 which returns one node for each match, the tree representation of XQuery can have one or more underlined nodes to identify output nodes. The results are a set of 3-tuples which consist of a-tagged, y-tagged and x-tagged node matches = $\{< r > a_1, y_1, x_1 < /r >, < r > a_5, y_4, x_4 < /r >, < r > a_5, y_5, x_4 < /r >\}$.

$for\ \$a\ in\ doc(\text{"}T_1\text{"})//a$
$for\ \$y\ in\ \$a/y$
$for\ \$x\ in\ \$a/x$
$return\ < r > \$a, \$y, \$x, < /r >$

(a)



(b)

Figure 2.11: An example of XQuery and its tree model

2.5 Conclusion

The purpose of this chapter is to outline the important aspects of XML that are relevant to this research study. In this chapter, XML databases and their supporting mechanisms have been described in Section 2.2, and the two core processes performed in XML databases: parsing and querying have been reviewed in Sections 2.3 and 2.4, respectively.

The next chapter presents the literature review of XML twig pattern query approaches. Several approaches and techniques previously proposed for processing XML structured queries will be investigated. The main goal of the following chapter is to identify the research problems which will be addressed later in Chapter 4.

Chapter 3

Related Work on XML Query Processing

3.1 Introduction

As enterprises and businesses produce and exchange XML-formatted information more frequently, consequently, there is an growing requirement for effective handling of queries on data which conforms to an XML format [146, 68, 201, 98, 81]. Recently, several approaches have been proposed in the literature to process XML queries [5, 147, 68, 38, 254, 6, 188, 101, 190, 147, 57, 139, 102, 211, 48, 249, 66, 112, 96, 53, 245, 70, 92, 240, 242, 127, 110, 146].

Effective matching of XML formatted tree patterns has been broadly considered as an essential operation in processing XML queries. Therefore, XML query processing has attracted a considerable number of researchers to improve the XML query processing performance. Proposed algorithms include TwigStack, TwigStackList, TJFast, Twig^{2}Stack, TwigList, TwigFast and TJStrictPre and GTPStack. Most of the state-of-the-art algorithms to evaluate XML queries will be outlined, discussed and surveyed in terms of critical parameters i.e, memory consumption, response time and I/O costs. This chapter aims to provide a general background to techniques used in XML query processing in Section 3.2. The research work related to XML query processing will be described in Section 3.3. The state-of-the-art approaches for processing XML queries will be discussed in detail as well.

3.2 XML Query Processing

An Extensible Markup Language (XML) tree pattern query (abbreviated as TPQ) usually can be symbolized as twig (referred to as a rooted-labelled tree) which is considered as one of the common structural queries. As discussed in Chapter 2, single path expressions are the simplest structure of XML queries, thus, research into it has a long history and simple path expressions are not in the interest of this study. On the other hand, branching path expressions are essential to the pattern language which is the core of most languages

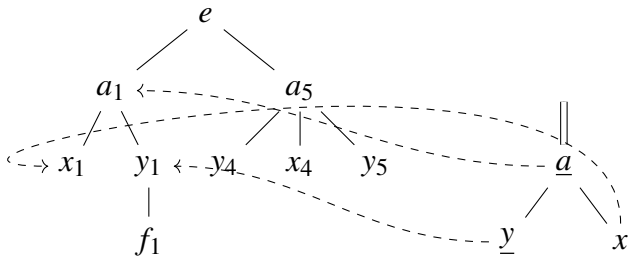
(i.e., XPath and its sub-languages) for processing XML documents [119]. The main aim of this study is to investigate efficient evaluation techniques for branching path expressions. In general, an XML query is defined as a complex selection on elements of an XML document specified by structural information of the selected elements. Improving the efficiency of twig patterns matching is a core operation in processing of an XML query [110, 147, 146, 144, 172, 146, 15, 53, 89, 87, 22] since tree patterns are the basis for querying structured tree-based data model such as XML.

There are two main types of XML query: a keyword-based query and a structured-oriented query. Keyword-based query comes from the traditional informational retrieval field [248, 26, 51, 124, 123]. In this category of query, an XML document is queried based on their textual contents without considering the semantics carried by the document structural information. This type of XML query is irrelevant of this research study but an overview of the work related to keyword-based queries will be described further in Section 3.2.2. By way of contrast, a structured query is the commonly used definition of a query which concerns not only the content but also the structure of XML documents, the effectiveness of queries in this group depends on users' knowledge of the document structure.

The underlying data model for XML documents is a labelled tree, where nodes are elements of XML documents and edges represent relationships between them, plays an important role in XML retrieval algorithms [20, 81, 155, 68, 40, 61]. The same representation is used for structured queries, as a result, the retrieval process can be seen as a matching problem between XML documents and XML query patterns [209, 146, 70].

One of the most important problems in XML query processing is tree pattern matching. Generally, tree pattern matching is defined as a mapping function M between a given tree pattern query Q and XML data D, $M : Q \rightarrow D$ that maps nodes of Q into nodes of D where structural relationships are preserved and the predicates of Q are satisfied. Formally, tree pattern matching is to find all matches of a given tree pattern query Q in an XML document D [81, 209, 93, 40, 5, 68, 146]. For a document D and a query Q with n nodes $(q_1, \ldots, q_n)$, a complete match is an n-dimensional tuple $(e_1, \ldots, e_n)$ that consists of the database elements that identify a distinct match of Q in D. On the other hand, an output match is a projection of a complete match such that the database elements corresponding to non-output query nodes are excluded [148]. The answer of Q on D is an ordered set of all the output matches of Q on D. The matching tuples have to be in the sorted order of the common prefixes of the individual root-to-leaf paths. Figure 3.1 shows XML data and a TPQ with a match.

The aim of this section is to outline the state-of-the-art techniques in addressing tree pattern query matching problems. An overview will be given to describe the different methodologies exploited in the field of XML query processing in the following subsections.



(a) an XML tree T_1 and a TPQ Q_1

a_1	y_1	x_1
a_5	y_4	x_4
a_5	y_5	x_4

(b) the complete match

a_1	y_1
a_5	y_4
a_5	y_5

(c) the output match

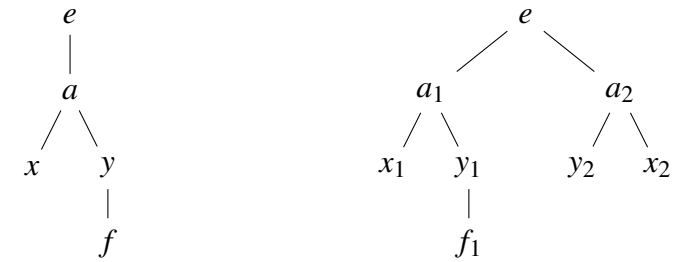
Figure 3.1: An example of XML tree T_1 and a TPQ Q_1 with matches. The output query nodes are underlined.

3.2.1 XML Indexing

Generally, the purpose of indexing is to improve the efficiency and the scalability of query processing by reducing the search space [98, 140, 174, 243, 250]. Without an index, retrieval algorithms have to scan all the data (in the context of XML, every node in an XML document), which would degrade the performance of retrieving processes by consuming a considerable amount of time and space. In XML technology, there are two basic approaches currently being adopted in research into XML indexes. One is the content index approach which indexes data values in XML documents (for instance, B+ tree indices) and the other is the structural index approach which indexes the structure of XML documents [1, 154, 140, 119, 52, 24]. The structural indexing approach may be further divided into three main classes, namely, path indexing, node indexing and sequence-based indexing. Many indexing methods based on semi-structured data have been proposed in the literature [4, 243, 229, 248, 254, 84, 60, 119, 52, 147, 85, 109, 124, 175, 184, 17, 10, 2].

Path indices, also known as structural summaries and index graph schemes, usually summarize all paths in an XML document starting from the root. A classic example of a path graph index is a *DataGuide* [85]. It is a typical model that merges all equivalent paths into a single one so that every path in a *DataGuide* is unique. One major drawback of this approach is that it only supports a simple path queries. An example of a *DataGuide* for a given XML tree is depicted in Figure 3.2a.

To tackle this problem, [240] proposed a new concise data structure, called *VersionTree*, to carry the structural information of the original XML document based upon the existing *DataGuide*, coupled with a covering index for branching path expressions. The new summarized tree supports efficient evaluation of twig queries and improves XML query processing performance. In contrast, the work of [254] addressed the indexing problems when storing XML in relational model. The researchers developed two index structures, namely: ROOTPATHS and DATAPATHS, which are effective for processing XML queries.



(a) DataGuide (b) *F&B*

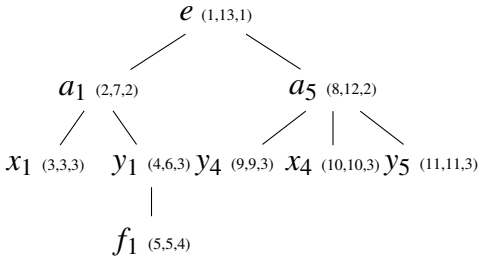
Figure 3.2: Structural summaries for the XML tree T_1 in Figure 3.1.

The proposals can seamlessly integrate with SQL query processors in relational database management systems. Another group of structural summaries rely on the idea of grouping nodes in an XML tree based on local similarity, called similarity-based indexes which are designed to support frequent path queries. When the index is able to adjust its structure according to a specific query work load, it is called an adaptive path index, a well-known example of this category is the *D(K)* where *K* is the length of the label path considered during grouping [52]. The *D(K)* has a major limitation that is the value of *K* influences the index performance. A covering index, *F&B* index, similar to that in RDBMSs, has been considered to support all branching path queries [119, 147]. The main limitation of a covering index, however, is that the index size tends to be very large, in most cases close to the size of the data tree. Figure 3.2b shows the *F&B* index for the data in Figure 3.1. Meanwhile, another approach was proposed in [102] to reduce the size of the summarized tree further by storing the structural information obtained into two *hash* tables. The proposed indexing method is called *CIS-x* and according to the experimental results, it yields a good performance in terms of index space and query efficiency. However, it is a memory-based index which makes it unable to handle large XML documents and it does not support updating. Similarly, [7] exploited the notion of objects of XML data in indexing all paths of the XML document. An object of an XML document is defined as a non-leaf element that consists of simple or other complex elements. Object-based XML data partitioning technique was introduced to structure large XML data logically for query processing. The authors proposed a series of indices, namely: schema index, data index and value index based on the utilization of structural and content components of XML data. The schema and data index are proposed to maintain the structural constraints of XML queries while the value index is used to improve the performance of querying constant values within XML data.

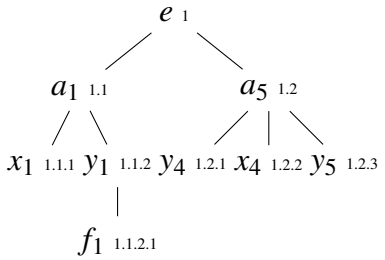
Node indexing approaches index each node in an XML document by assigning an unique label (based on a labelling scheme) to every node which records its positional information within an XML tree. The values of labels vary according to the chosen labelling scheme. Moreover, this group of indices utilizes nodes as the basic unit of a query which provides a great flexibility in performing any structural query efficiently by

the matching nodes of a query via structural joins. Using a node index is also known as a labelling scheme, numbering scheme or node encoding. According to [154], a labelling scheme has to guarantee uniqueness and order preservation of node labels, thus the hierarchical relationships between a pair of nodes can be determined efficiently. The labelling scheme should enable checking all XPath relationships by computations only. To better understand the mechanisms of node indexing methods and their properties, [98] classified node indexing into four distinct types; Sub-tree labelling, Prefix-based labelling, Multiplicative labelling and Hybrid labelling. However, a full discussion of the different categories lies beyond the scope of this study. Generally, all XML query processing algorithms which perform structural join operations to match a given query against an XML document rely on either sub-tree labelling schemes or prefix-based labelling schemes [40, 147, 146, 5, 236, 19, 140, 138, 184, 205]. A well-known example of sub-tree labelling is the regional labelling scheme proposed in [253]. In this approach, each node is assigned with a 3-tuple as $< start, end, level >$. *Start* and *end* contain values of positions corresponding to the opening tag $< tag >$ and the closing tag $< /tag >$. Level represents the depth of a node within an XML tree. The two basic relationships Ancestor-Descendant (A-D) and Parent-Child (P-C) can be determined efficiently. Given two nodes u and v, u is an ancestor of v if and only if $u.start < v.start < v.end < u.end$. Furthermore, a P-C relationship is defined as node u is the parent of node v if and only if $u.start < v.star < v.end < u.end$, $v.level = u.level + 1$. By way of explanation, node v is in the range of node u. A notable example of prefix-based labelling (also known as a path-based labelling scheme) is the Dewey labelling scheme [243]. In the Dewey labelling scheme each node is associated with a sequence of integers that represents the node-ID path from the root to the node. The sequence of components in a Dewey label is separated by "." where the last component is called the self label (i.e, the local order of the node) and the rest of the components is called the parent label. For instance, consider the XML tree in Figure 3.1, Figure 3.3 presents labelling an XML tree using three different labelling schemes. Using the Dewey labelling scheme, given two nodes u and v, u is an ancestor of v if and only if *u.label* is a prefix of *v.label*. For instance, a_1 is the ancestor of f_1 because $\{1.1\}$ is a prefix of $\{1.1.2.1\}$ as shown in Figure 3.3. In the same way, $\{1.1\}$ is the parent of $\{1.1.2\}$ because path-based labelling schemes encode the P–C relationship by extending the parent's label with a component for the child. The main disadvantage of these two labelling schemes is that they fail to take XML documents with frequent updates into consideration. Thus, updating can incur a heavy cost of re-labelling. Another problem with prefix-based labelling schemes is that the performance of determining structural relationships is affected by the depth of the XML tree due to the increase in the size of labels.

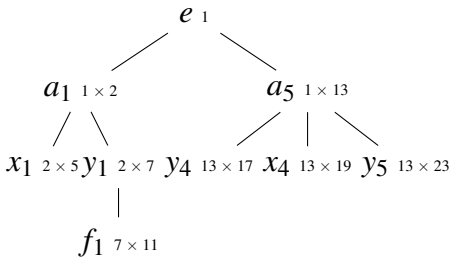
More recently, [147] proposed a new labelling scheme based on Dewey to speed up the evaluation of XML twig queries. The proposed labelling scheme has the ability to derive a unique path from the root to a given node and maps a given node label to a series of element names. Also, it can be extended to handle dynamic documents and avoid re-labelling. One



(a) regional (range-based) labelling scheme



(b) Dewey labelling scheme



(c) prime number labelling scheme (top-down approach)

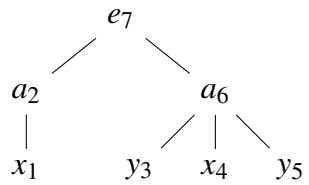
Figure 3.3: labelling schemes for the XML tree in Figure 3.1.

study by [243] examined the trend in designing an effective labelling scheme for both static and dynamic XML documents. The researchers proposed a novel labelling scheme which is a variant of Dewey, called dynamic Dewey encoding (abbreviated as DDE). A novel ordering concept was defined to transform the original Dewey into a fully dynamic labelling scheme; thus re-labelling can be avoided. DDE support high query performance and has the ability to avoid re-labelling completely. [84] exploited the notion of IP addressing and sub-netting techniques in computer networks to propose a novel labelling scheme. The proposed scheme is efficient in determining the hierarchical relationships between two given nodes very quickly by applying a simple logical AND operation. The new scheme is called XDAS and shows superior efficiency in terms of reducing label size. [229] also proposed a new numbering scheme to support dynamic XML documents and avoid re-labelling. The new labelling scheme shows a space-efficiency in terms of the length of labels and time consumed to update XML documents. Another approach was proposed in [248] to improve content-oriented query (keyword search). The proposed labelling scheme was coupled with a inverted index on the content nodes to provide rapid access. Likewise, focusing on labelling dynamic XML documents, [244] proposed a new encoding technique that can completely avoid re-labelling when updating occurs. The proposed encoding technique utilizes vector-based order to label nodes instead of natural or lexicographical order. Based on the encoding technique mention above, [174] studied the influence of applying vector-based order on some of the existing labelling schemes. Vector-based order is a novel order concept to avoid relabelling when XML documents change frequently. A vector code is a 2-tuple of the form (x, y) where x and y are integers with $y \neq 0$. Given two vector codes A:(x1, y1) and B:(x2, y2) A precedes B in vector

preorder if and only if and only if x1/y1 $<$ x2/y2. The experimental results indicate that using vector-based order reduces the size of the label and improves the performance of the query system. In addition, [4] reviewed, evaluated and matched several of dynamic labelling schemes that support dynamic XML documents. Particularly, the researchers considered various labelling schemes that depend on the Dewey encoding concept. Such labelling schemes are preferable to containment labelling schemes when it comes update competence; but, these schemes are considered expensive in terms of storage. Furthermore, a novel labelling scheme was proposed, a sibling-labelling scheme which depends on the notable Dewey coding concept. By utilizing the proposed scheme merely two nodes need to be re-labelled. However, sibling-labelling scheme is very efficient in supporting the common axes (Parent-Child, Ancestor-Descendant, siblings relationships and document order) and restoring the relations among the given nodes. In the same way, the authors in [138] proposed a novel dynamic prefixed labelling scheme (DPLS) which can re-use labels deleted in order to reduce the label size. A classic example of multiplicative labelling is a prime number labelling scheme (see Figure 3.3) which was proposed to support labelling dynamic XML documents. In a prime number labelling scheme [238], every node is given a unique prime number called the self-label. Then the label for each node is the product of its self-label and its parent-label. This labelling scheme completely avoids re-labelling when a new node is inserted, only simultaneous congruence value to determine the document order needs to be recalculated. In [2], a parallel algorithm for prime number labelling scheme was proposed so that the prime number labelling scheme can deal with huge XML documents. More recently, a new labelling scheme which combines the advantages of range-based and prefix-based schemes was proposed in [186]. The new labelling scheme is an extension to the prime number labelling scheme introduced in [238]. The basic idea is to encode the structural information using prime numbers similar to that used in [238] and the middle fraction for encoding the document order so that there is no need to recalculate simultaneous congruence value. A combination of structural summary and prime labelling scheme has been proposed in [163] to estimate the result size of a TPQ.

Finally, sequence-based indexing methods are used to transform both XML documents and queries into sequences. Tree pattern matching then is reduced to subsequence matching [98, 146, 137, 96, 236]. Early examples of research into sequence-based indexing include [188, 157, 230, 213]. Sequence indexing approaches suffer from two problems, namely, false positives (also known as false alarms or imprecise result) and false negatives (also known as false dismissals or incomplete result). Time-consuming postprocessing is necessary to overcome these problems and identify the actual result from the subsequence matchings. In [188], based on *Prüfer* codes, XML documents and queries are transformed into sequences where an one-to-one correspondence between trees and sequences are formed. The most outstanding feature of the proposed indexing method is that it can prevent false positives caused by transforming the hierarchical structure of XML documents into equivalent sequences when using a structure-encoded sequence introduced in the previous

work of [230]. However, this method of indexing has a number of limitations. Approaches, [188, 213], in this category require a large amount of computation, called refinement phases, which are performed after the matching phase to avoid the generation of false negatives which occur when a branching query node has multiple identical child nodes. This will be explained more in Section 3.3.2.3. Figures 3.4 illustrates the process of transforming an XML document and an XML query into sequences in PRIX (*PRüfer* sequences for Indexing XML) introduced in [188].



(a) an XML tree T_2 with nodes associated post-order numbering

Number *prüfer* sequence:	{2,7,6}
Label Sequence	$\{a_2, e_7, a_6\}$

(b)

Figure 3.4: An illustration to PRIX.

Overall, these studies highlight the need for XML indexing methods in XML retrieval systems and how these indexing methods play an important role in improving the XML query processing performance by reducing the search space and providing the structural relationships between nodes within XML documents. All the studies reviewed so far, however, suffer from a large index size, long construction time and an absence of some structural information. These are still open issues in the field of indexing XML documents.

3.2.2 XML Keyword Search

XML keyword search queries, also called information retrieval style queries, generally do not have structural parts and consist only of the content parts specified as a set of keywords $k = \{k_1,...,k_n\}$. The answers to keyword queries in XML are sub-trees of XML documents which contain all the query keywords. The sub-trees returned by a keyword query are computed by finding the lowest common ancestors (LCA for short) of all query keywords, the lowest common ancestors defines the group of elements that are likely to be relevant to a query [26, 51, 248, 142]. The most commonly used of Moreover, a ranking function is required to identify the most appropriate candidate query results based on their relevance to keyword queries, only high ranked sub-trees are returned to users.

The authors of [26] addressed the problems of users' intentions in keyword queries and ranking results. The researchers proposed a new approach to analyse XML documents statistically to determine the appearances of keywords within XML documents either as tag names or as text values. Also, a new ranking method based on the statistical analysis of the XML documents was proposed. Concentrating on improving the efficiency of the top-k keyword search queries which is a set of K elements (or nodes) with the highest ranking relevant to keywords searched, [51] proposed a novel node labelling scheme, called

JDewey that preserves the order of all nodes occurring at the same level. A bottom-up processing algorithm was devised to compute efficiently the smallest, lowest common ancestor which is a variant of the lowest common ancestor. The smallest lowest common ancestor, or SLCA contains all keywords of a query but has no sub-tree which also contains all the keywords. In other words, the closest internal node that contains all keywords of a given query.

3.3 Tree Matching

Extensible Mark-up Language (XML) has become the standard data representation format for exchanging many types of data. XML documents may be classified on the basis of their content into data-oriented documents and text-oriented documents. XML documents of the first category are highly structured and commonly stored in databases, text-oriented documents have an irregular structure [72, 6, 110, 31, 98, 209]. The growing number of XML documents leads to the need for appropriate XML querying algorithms which are able to utilize the specific characteristics of XML documents. In these documents, data is hierarchically structured as are the queries and evaluation is performed using tree matching methods. In order to state the tree matching problem, in addition to the definition in Section 3.2, a labelled tree is defined as $T = (V,E,r,\mu)$, where $V = (v_1,...,v_n)$ is a finite set of nodes, $E = \{(u,v) \in V \times V\}$ is a set of edges. $r \in V$ is the root node and μ is a labelling function which maps each node in T to one label in a finite set of labels $L = (l_1,..,l_n)$. In a labelled ordered tree, the function call level $l(u)$ of node u is defined as the number of distinct edge(s) from the root to u. Therefore, tree pattern match problem between a given tree $T_1 = (V_1,E_1,r_1,\mu_1)$ and a given tree pattern query $P = T_2 = (V_2,E_2,r_2,\mu_2)$ is defined as a mapping function (as discussed in Section 3.2) from the nodes of T_2 into the nodes of T_1 such that the following must be satisfied [209, 110, 31, 15]:

1. *if* $v_2 \in V_2$ *maps to* $v_1 \in V_1$, *then* $\mu(v_1) = \mu(v_2)$,

2. *if* $v_2 \in V_2$ *maps to* $v_1 \in V_1$ *and* v_1 *is an internal node (not leaf node), then each child of* v_2 *maps to at least one child of* v_1.

In the context of XML, tree pattern matching can be grouped into two classes: exact matching or approximate matching. Unlike exact matching where finding all occurrences of a given twig on a given document is necessary, the approximate matching technique aims at measuring the similarity between two given trees and quickly returns approximate answers to XML queries. The major difference between *exact* and *approximate* matching is that the restrictions on the mapping function which is defined above in Section 3.2 are required in *exact* matching which is strict in terms of structural relationships and the predicates which must be satisfied, while *approximate* matching is not.

Before proceeding to examine tree matching, it will be necessary to recall that XML query processing approaches may be divided into two main sub-groups. The relational approaches which store XML documents in relational databases and transform twig queries

over XML documents to SQL queries over relational tables and the native approaches in which storage and query processing mechanisms are built from scratch without involving relational databases. Only the native approaches are relevant to this thesis, although several studies investigating the relational approaches have been carried out [59, 3, 254, 211, 253, 98, 245, 236, 204, 197, 187]. So far this chapter has considered XML query processing as a tree pattern matching problem. The following sections will discuss approximate matching approaches in Section 3.3.1 and exact matching in Section 3.3.2.

3.3.1 Approximate Matching

This section only gives an overview of approximate matching because it is irrelevant to the work of this study. Approximate tree pattern matching is the technique of determining approximately the best match of one tree against another [129]. In the field of XML query processing, approximate tree pattern matching is the process of finding a given tree pattern query Q that matches a given an XML document D approximately rather than exactly.

The goal of this technique is to increase users' satisfaction rather than matching efficiency [93, 31, 135]. Approximate matching techniques actually summarise by measuring the similarity between two given trees utilizing the minimum possible sequence of operations to get the two trees to fit into each other. The issue with this approach is attempting to find one match between a twig pattern and a target tree, whereas in reality tree matching for XML retrieval systems requires the output of all possible matching, also the existence of A-D relationships in twig pattern queries makes the matching problem more complex [212, 40, 155, 129, 1]. Despite the limitations of approximate matching for XML retrieval systems, an approximate matching process such as query rewriting, query relaxation and query expansion has been utilized to answer XML queries. Such a process is not relevant to this thesis.

The process of tree matching is exploited in the field of XML clustering by dividing a large collection of XML documents into groups according to their similar characteristics (i.e, structural, content and semantic similarity). One study by [31] examined the trend in tree edit distance and according to the researcher, tree edit distance is the first approach proposed to find the similarity between two trees by computing the minimum number of operations to transform the former tree in the latter one, these operations are: node insertion, node deletion and node re-labelling. The issue with this approach is that it is computational expensive and makes it impossible to measure the structural similarity information. Another approach has been advanced to overcome these problems, [129] proposed a new algorithm to improve approximate join quality by considering both XML structure and node labels. In [167], the authors proposed a new methodology to determine the similarity between heterogeneous XML schemas which are modelled as trees by considering semantic as well as the hierarchical similarity of the elements. The quality of match model has been proposed in [62] to match two XML schemas and approximate matching has been considered in this model by quantifying the match according a measure

of some degree of match called relaxed match. Interestingly, the work of [212] transformed the tree-based structure of XML data into a multi-set of pivotal elements, wherein each pivot (i.e, set) contains the labels of two nodes and their lowest common ancestor (LCA). Hence, the similarity between two trees can be reduced to the similarity between their encoded pivots. Based on the concept of edit tree operations, [214] has addressed the lack of utilizing the structural similarities between sub-trees to compute the similarity between XML documents.

Even though, approximate matching for XML retrieval systems and their relevant techniques are out of the scope of this thesis, an overview has been given in this section because they are widely used in developing XML query processing system as filtration phase of XML documents resembles to that in [9]. The next section will outline exact matching approaches for XML query processing.

3.3.2 Exact Matching

Exact matching is a process which allows users to retrieve the information when a tree pattern query matches exactly, according to the mapping function defined in Section 3.2, data stored within databases. As an illustration, the exact matching process as mapping function between two trees is shown in Figure 3.5. Exact matching for processing XML queries has been well studied since one of the main usages of tree patterns is to express and optimize queries over tree-based structured data [15, 93, 24]. Several attempts have been made in the literature to improve XML retrieval systems based on the exact matching approach [105, 91, 253, 5, 127, 171, 53, 188, 213, 230, 128, 19, 136, 255, 25, 96, 89, 132, 130, 206, 70, 217, 102, 181, 125, 172, 67, 22, 87]. The process of querying an XML document is to identify a specific data pattern in the given document by using an XML query language such as XPath or XQuery. Due to the importance of twig tree pattern queries in XML retrieval systems, finding all matching occurrences of a tree pattern query in an XML document is often considered as a specific task for XML databases as well as a core operation in XML query processing.

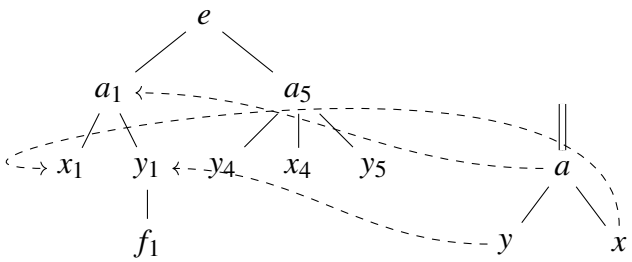


Figure 3.5: An example of exact matching process (exact mapping).

Exact matching approaches may be further classified into three main types according to the type of the indexing methods exploited as discussed above in Section 3.2.1. These types are binary Structural Join, holistic structural join and sequence-based approaches.

Algorithms of the first and the second groups usually use an inverted index mechanism wherein labels for each distinct tag name are stored in one inverted list (also referred to as a streaming list or a label list) and ordered based on the order of their appearance in the document, or the document order. For instance, Figure 3.6 illustrates the inverted lists for the XML data tree in Figure 3.3. Most algorithms in this category rely on nodes indexing methods (i.e, labelling scheme) to capture the common axes in XML query languages and facilitate XML query processing. That is, labels retrieved for each query node tag are merged by twig matching algorithms such as TwigStack to return the result of a TPQ. Using streaming lists refer to partition index (i.e., the XML data is partitioned to streams where the input data is read once) which uses the XML node tag as a key while document index uses the node label as a key and the XML node tag as value in addition to some other information about the node [24].

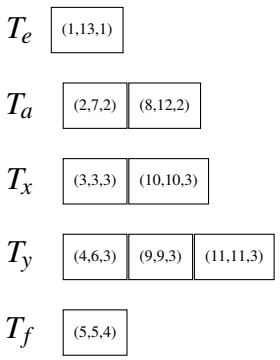


Figure 3.6: Streams containing range-based labels for five distinct tags in the XML tree in Figure 3.3.

This section has reviewed the three key aspects of exact matching for XML retrieval approaches. The next subsections will discuss the methods used in these approaches. The first section 3.3.2.1 will outline the binary structural joins. Then, the holistic join-based approaches will be investigated in Section 3.3.2.2. Finally, the sequence-based approaches will be described in section 3.3.2.3.

3.3.2.1 Binary Structural Join Approaches

The basic structural relationships between nodes in tree-based models are the Parent-Child and Ancestor-Descendant. A core operation in XML query processing is searching all the occurrences of these relationships. The pioneering work of [253] is the first native approach that developed binary structural joins. The authors introduce a special inverted list technique similar to that in Figure 3.6 based on the regional labelling scheme where each node is associated with triplets with a *(start,end,level)* label (Section 3.2.1). They proposed a structural join algorithm (i.e., specialised loop joins) to find all the occurrences of the basic transitive closures of the data tree (i.e., the basic axes "/" and "//" in XPath). The proposed algorithm benefits from the notion of a labelling scheme in which all positional information

about data elements are recorded in labels and has shown a significant improvement over the traditional merge join algorithm utilized in RDBMS.

This approach is called a binary structural approach because it performs a binary join where the input is two sets of nodes to be joined to eliminate irrelevant nodes, then the output of this join may be used as the input of another join. Binary join approaches, from their names, only join two sets at the same time, in other words, they only join two label lists at a time. The major drawback of this approach is that the I/O cost is very high due to the fact that a query node may be accessed several times at any point during execution. To tackle this limitation, the Stack-tree algorithm was proposed in [5] to avoid scanning query nodes more than once. The researchers augmented the structural join algorithm with a stack or LIFO (Last In First Out) data type such that a single sequential scan for the two sets of nodes would capture all the basic relationships without any recursive scan. Algorithms of this binary join family are based on a decomposition technique that decomposes a given query into a set of binary relationships, then performs structural joins to generate intermediate results. Finally, the query answer is produced by stitching the intermediate results together. The size of intermediate results has a significant impact on XML query processing performance. The researchers in [207] provided a new solution for efficiently processing queries with sibling axes using a sibling list for each level in the XML tree. In the same way, [217] introduced a new technique which can provide an efficient solution for processing positional predicates in binary structural join approaches.

[40] extended the Stack-tree algorithm by utilizing a stack for each query node in a single path from root-to-leaf. Their algorithm is called PathStack and the encoding representation of query nodes results in intermediate results of manageable size compared to the input and the output lists. Figure 3.7 illustrates the stack encoding of the PathStack algorithm. Elements in stacks are linked by pointers to produce the query answers without the need to decompose it into a set of binary relationships. PathStack uses a filtering mechanism to avoid pushing irrelevant nodes in the corresponding stacks. The downside of this algorithm is that it only supports a simple path query efficiently (Section 2.4). In order to have path matches in a sorted order of the common prefixes of the individual root-to-leaf paths, *self-list* and *inherit-list* are maintained for each stack item corresponding to an inner query node during the query evaluation. As a result, output matches are delayed until the algorithm ensures that there is no match prior to them in the sort order that should be outputted. The *self-list* and *inherit-list* for the XML data and query in Figure 3.7 are presented in Figure 3.8. The algorithm outputs the contents of the *self-list* and then the contents of the *inherit-list* to return paths in sorted root-to-leaf order. The extension of PathStack to support TPQs will be discussed in Section 3.3.2.2.

The authors of [127] have developed a native XML storage and query system, or XSQS for short. They have addressed the challenge of the size of intermediate results generated by Stack-tree algorithm and have proposed a new algorithm as an extension of Stack-tree called modified Stack-tree. According to the experimental results, the proposed algorithm

has better performance in terms of reducing the size of intermediate results. In an analysis of binary structural approaches, [148] found that the binary structural join algorithms can have linear time complexity with respect to the size of input and output for TPQs with a higher ratio of non-output query nodes when a fully pipelined (FP) plan is selected. In other words, the same complexity as the holistic twig join approaches but the difference is that the complexity of holistic approaches is not influenced by the number of non-output query nodes. Following the semantics of XQuery, output query nodes corresponding to 'return clauses in XQuery while non-output query nodes corresponding to all other query nodes. The FP plan guarantees no-blocking so that each join operation does not have to wait for the complete result of the previous join [241].

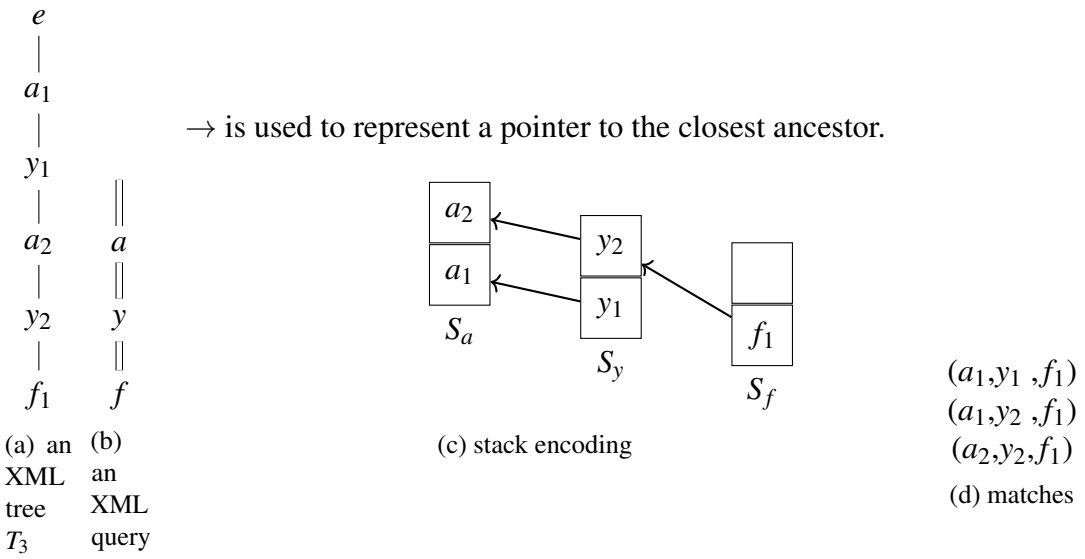


Figure 3.7: Illustration to PathStack.

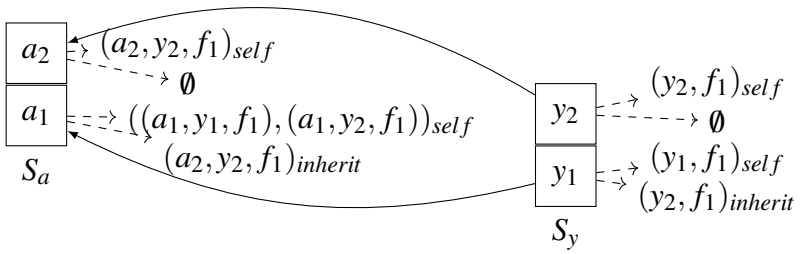


Figure 3.8: Stack items with final self-list and inherit-list for the XML data and query in Figure 3.7.

Although binary structural join approaches significantly degrade XML query processing performance, most XML algebras representing an XML query language by a logical query plan support binary structural joins [108, 83, 20, 24, 148]. However, an accurate query optimiser is an important parameter for the efficiency of binary structural joins. In the following section, the main algorithms in the holistic join-based approach are discussed.

3.3.2.2 Holistic Structural Join Approaches

The holistic join was introduced by [40] as a new approach to evaluate query twig patterns efficiently. The work was an extension to the sophisticated PathStack algorithm to support twig queries without decomposing the queries into a set of binary structural relationships. They proposed decomposing twigs into a set of root-to-leaf paths and the evaluation is performed for each root-to-leaf path using the PathStack algorithm. The final results are produced by a merge join operations for intermediate results generated by PathStack evaluation on decomposing single paths. The developed algorithm is called TwigStack and has shown a significant performance improvement in reducing intermediate results in comparison to the binary structural join algorithms. The TwigStack algorithm only guarantees an optimal evaluation of twig queries with Ancestor-Descendant relationships connecting all query nodes. The optimal evaluation in an holistic approach means every query node pushed into the encoding data structure types (in the case of TwigStack a chain of stacks) must be part of the final results by scanning them sequentially once [61]. However, TwigStack's performance suffers from generating useless intermediate results when twig queries encounter Parent-Child relationships. In [40], the XB-tree index, which is a variant of a B-tree index, was proposed to reduce the disk-read costs of TwigStack by skipping over input streams corresponding to inner query nodes which do not satisfy A-D relationships with child query nodes.

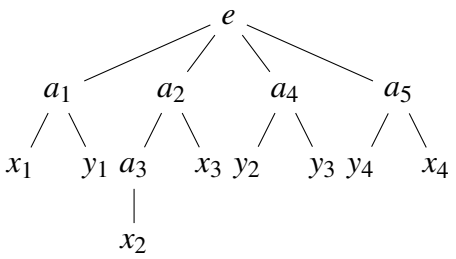
In general, TwigStack performs twig evaluation in two phases: the first phase is to decompose a twig pattern query into single root-to-leaf paths and then it matches them against XML data. The second phase is the merge phase in which all matching results produced by the first phase are merged to compute the final query answers. Obviously, the second phase is an expensive process since n-way merge has to be performed where n is the number of single paths in the twig query. TwigStack is considered as the keystone for algorithms using this approach, many research papers suggested improvement to the original TwigStack as in [53, 146, 147, 18, 9, 245, 249, 114, 111, 50, 116, 89, 54, 144, 18, 236, 118, 185, 240, 137, 145, 237, 47, 252]. Nevertheless, an optimal evaluation of tree pattern query with any arbitrary combination of Ancestor-Descendant and Parent-Child relationships has been proven to be impossible by [61] for the TwigStack algorithm and its variants. [198] researched the space complexity of XML twig queries over indexed documents for three different modes of query evaluation: tree matching, full-fledged and filtering queries. The authors have analytically proven the difficulty evaluating twig queries with Parent-Child edges in which large sets of unused intermediate results remain in the main memory. However, the worst case space complexity of twig pattern matching is an open question, the known bounds are $\Omega(max(d,u))$ and $O(I)$, where d is the longest path in the XML tree, u is the total number of elements which are part of a complete match and I is the size of the input tree [88].

XR-tree and TSGeneric^{+} were proposed in [114, 111], respectively. They were designed to speed up reading the input lists during the structural join operation in TwigStack.

XML Region Tree (XR-tree as abbreviation) was introduced based on regional labelling scheme to index XML nested elements in which all ancestors (parents) or descendants for a given elements can be obtained in an optimal time. The two research papers focused on improving scanning elements in label lists to avoid sequential scanning by jumping over irrelevant elements. The XR-tree index can be easily integrated into any holistic join algorithm. [77] extended the work of [111] to reduce the number of physical moves over the streams. This was achieved by applying virtual moves as much as possible. In order to avoid as many data structure reads as possible, nodes are forwarded to "virtual positions", which have only start values. Then, the query is processed bottom-up and top-down. In the bottom-up phase, nodes are forwarded to contain their descendants, and in the top-down phase, nodes are forwarded until they are contained by their parents. Eventually, the node with the minimal current start value is forwarded to a real data node. The authors in [130] introduced three optimization rules to improve the efficiency of the existing holistic twig matching algorithms. The basic idea of their algorithm, TJEssential, is to avoid unnecessary self-nested matching checks, the order in which child query nodes are checked and avoid unnecessary recursive calls when all elements in a stream have been scanned.

In [54], the authors reviewed the sub-optimality of the existing clustering technique used in TwigStack where an XML document is clustered into tag streams which group together elements with the same tag name. They proposed two novel different streaming schemes, namely: *prefix path* and *tag+level* streaming schemes. A *tag+level* streaming scheme contains all elements which have the same tag and located in the same level. A prefix-path streaming scheme, or PPS for short is ordered set of elements which have the same prefix path. Based on the introduced streaming schemes, they proposed an extension to TwigStack called iTwigJoin. Their algorithm is optimal for queries with A-D edges only when tag streaming schemes (i.e, label lists) is applied, the utilization of *tag+level* streaming scheme in iTwigJoin guarantees the optimality in two classes of queries: A-D or P-C edges only. In addition, the iTwigJoin depends on *prefix path* (i.e, iTwigJoin+PPS) is optimal in three classes of queries: A-D, P-C edges or one branching node only. It has been proven that the efficiency of iTwigJoin can be reduced when the number of streams for every query node is increased. Figure 3.9 presents different streaming schemes over an XML tree, elements are clustered based on similarity in tag and level and the number associated with each tag indicating the level. For example, x^3 streaming list contains all elements with x-tagged node and appear in level 3. A recursive prefix path streaming scheme was proposed in [50] to support highly recursive XML data. The new streaming scheme is an extension of prefix path scheme to reduce I/O cost when a recursive prefix path occurs on particular tags. The authors in [21] introduced a new holistic approach for searching prefix path streams efficiently instead of the dynamic programming used in [54]. The basic idea of their approach is to retrieve the set of matched labelled paths (i.e., matching streams) by finding nodes corresponding to those labelled paths in a *DataGuide*.

The experimental results have proven that the solutions based on holistic approaches are more robust than the stream pruning technique presented in [54].



(a) an XML tree

e:$\{e\}$
a:$\{a_1, a_2, a_3, a_4, a_5\}$
x:$\{x_1, x_2, x_3, x_4\}$
y:$\{y_1, y_2, y_3, y_4\}$

(b) tag streams

e^1:$\{e\}$
a^2:$\{a_1, a_2, a_4, a_5\}$
a^3:$\{a_3\}$
x^3:$\{x_1, x_3, x_4\}$
x^4:$\{x_2\}$
y^3:$\{y_1, y_2, y_3, y_4\}$

(c) tag+level streams

/e:$\{e\}$
/e/a:$\{a_1, a_2, a_4, a_5\}$
/e/a/a:$\{a_3\}$
/e/a/x:$\{x_1, x_3, x_4\}$
/e/a/a/x:$\{x_2\}$
/e/a/y:$\{y_1, y_2, y_3, y_4\}$

(d) prefix path streams

Figure 3.9: Illustration of different partitioning schemes.

The researchers in [18] introduced a variant of iTwigJoin+PPS called TwigStackSort which reduced the time to search for the equivalent set of prefix-path streams participant in a given twig query by searching a *DataGuide* first and using the binary search algorithm to retrieve elements with the minimal start value during query processing. A set of experiments was conducted to verify the correctness of their algorithm and has shown that the number of labelled paths can degrade the efficiency of algorithms based on prefix label streaming scheme. Similarly, the same authors proposed a new holistic algorithm utilizing prefix path labelling scheme (in this case Dewey labelling scheme) called TJDewey in [19].

An early work to reduce the number of intermediate path solutions generated by TwigStack when twig patterns contain Parent-Child edges has been introduced in [144]. The researchers proposed a new algorithm TwigStackList that utilizes two data structure types for every query node in a twig query: stack as in TwigStack and a list. The key idea is to buffer some elements with P-C in lists to eliminate redundant path solutions. Subsequently, TwigStackList ensures optimal CPU and I/O costs when twig queries contain only Ancestor-Descendant edges below branching nodes and allows the occurrences of Parent-Child elsewhere. However, TwigStackList performs multiple scans of elements in the buffering lists so that relevant parents can be returned to the main algorithm. TwigBuffer [131] extended TwigStackList by using a complex buffering technique in order to avoid the generation of useless intermediate paths. Unlike the previous algorithms, TwigBuffer does not guarantee that elements in stacks (from bottom to top) do not lie on a root-to-leaf

path in the XML document. The main disadvantage of TwigStackList and TwigBuffer is that output matches are not presented in sorted root-to-leaf order. More recently, a new algorithm, QTwig was proposed in [206]. The QTwig algorithm is based on a new labelling scheme, ReLab^{+}, which is an extension of the range-based labelling scheme. That is, each node in the XML document is associated with 3-tuple as $< self, region, parent >$. *Self* is a unique number assigned to each node by incorporating the preorder traversal ordinal numbers into the nodes. *Region* is the *self* value of the right-most node of a subtree rooted from the current node. The parent attribute is the *self* field of the parent node. QTwig (Quick-Twig) is an extension of TwigStack for checking P-C edges before storing nodes in their corresponding stacks using *parentList* which is a *hash* table built during the query processing. The *parentList* table stores the self attribute as a key and the parent field as value. However, the authors overlooked the fact that TwigStack checks P-C relationships before pushing nodes into stacks by inspecting top elements [89] (This issue will be discussed in Chapter 3).

An alternative approach, using pre-processing filtration phases was introduced in [9] to reduce the search space. The horizontal filtration phase is based on an element labelling scheme (containment labelling scheme) to prune irrelevant nodes. The vertical filtration is based on a binary labelling scheme to compute the nearest common ancestor in constant time. The final step is to construct the structure matching in a bottom-up fashion. The proposed technique, TWIX, is effective when a query contains keywords in leaf nodes (selective queries) and only works for data-oriented XML documents. Their techniques are tailored to a specific class of XML documents such as DBLP (i.e., highly structured XML document). A wide range of XML datasets will be discussed in Chapter 5. Improving their techniques to make these mechanisms general to a wider class of XML documents is not a trivial task. In addition, the researchers in [199] highlighted the need to consider the XML node tags during the labelling process in order to improve the efficiency of XML query processing. They used a binary encoding to record child tags based on the information obtained from the XML schema. They introduced a new approach which requires a pre-processing step to filter out useless elements from the streams. Such an approach, however, has failed to satisfy the property of holistic approaches because the input streams corresponding to parent query nodes are scanned twice and stored in the main memory. It seems possible that these results are due to the complexity of binary encoding during the query evaluation. The paper made no attempt to give sufficient consideration to an XML dataset which has irregular structure and many recursive tags such as TreeBank. Furthermore, binary encoding is too expensive to be used for recursive XML documents [186].

To support content search, [236] developed a new algorithm to speed up the process of XML twig pattern matching by introducing the object and property concepts to holistic twig joins. The property of such value in an XML tree is the parent node of that value node. Then, the corresponding object can be considered as the ancestor/parent node of

each property. In their approach, non-value nodes are transformed into inverted lists and value nodes are stored in relational storage. The researchers justified reordering the steps of the previous holistic join approaches where the structural matching is performed first, then the value matching is executed. Accordingly, the new order results in high selectivity. The new approach, TwigTable, is a combination of TwigStack and SQL processing. That is, TwigStack is used to perform the structural matching while SQL processing is performed to reduce the input lists when TPQs contain value constraints. A group of experimental results has proven that TwigTable outperforms TwigStack when twig pattern queries contain content-search constraints, otherwise they have the same performance. Although, TwigTable has improved the XML query processing in terms of response time and I/O overhead, it violates the feature of native XML databases because it relies on RDBMS to store value nodes and on SQL processor to perform value comparison operations.

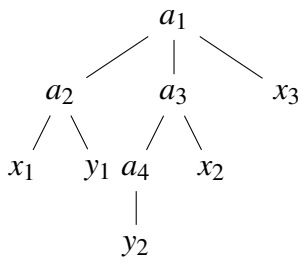
All the algorithms mentioned previously are based on decomposing twig queries into individual root-to-leaf paths and process queries in a top-down manner. The top-down process, in TwigStack and its variants, is a filtering strategy to eliminate irrelevant nodes which match query nodes' tags but do not satisfy structural constraints specified in queries. The top-down filtering can be seen as prefix path matching where a sequence of steps in XPath expression connects descendants to their ancestors. For example, consider a given query consisting of k query nodes as $q_1/q_2/.../q_k$ and if document element e corresponding to q_2 in the mapping function $q_2 \rightarrow e$. If and only if e has a parent element corresponding to q_1 which is satisfied the mapping function and so on to the leaf query node q_k, then each element in the entire path will be pushed into their corresponding stacks. In other words, the top-down process checks document elements in pre-order and stores them in the representation data structure in post-order.

However, the approach to examine XML queries against document elements in post-order was first introduced by [53]. The authors have proven in their paper that decomposition of twigs into a set of single paths and enumeration of these paths are not necessary to process twig pattern queries. The key idea of their approach is based on the proposition that when visiting document elements in post-order (i.e, reversed order) any element e the determination whether or not e satisfies the twig query sub-rooted at e can be drawn directly without further investigation because all its descendants have been visited. They proposed a new algorithm to process twig queries without merge joining single paths. A new encoding representation was introduced to store twig results in main memory. Unlike TwigStack where every query node has a stack to represent intermediate results, they proposed a tree of stacks in which every query node n is associated with a hierarchical stack HS_n which consists of an ordered sequence of stack trees, where each tree node is a stack. Pointers are heavily used to capture the basic relationships between elements in different hierarchical stacks as shown in Figure 3.10. The researchers developed a new algorithm called Twig^{2}Stack to evaluate a wider range of XML queries including generalized twig pattern (GTP) queries which contain both mandatory and optional relationships.

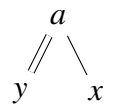
The mandatory relationships correspond to those path expressions in the FOR or WHERE clauses. The optional relationships correspond to those path expressions in the LET clauses. For a given GTP which is a fundamental building block for XQuery processing, not all nodes are return nodes. For the path expressions in the FOR clause, only the last node is the return node. [58]. Twig^{2}Stack produces the eventual answers by performing an enumeration function using the pointers in the hierarchical stacks. In the same context, a new algorithm was proposed in [116], called HolisticTwigStack. The new algorithm introduced the filtering strategy of TwigStack to Twig^{2}Stack in order to reduce memory consumption. The major limitations with these two algorithms is that the time taken to maintain the complex stack structure is significant. Even though, they both reduce the cost of query execution by eliminating the merge phase (second phase in TwigStack). In the worst case the entire document needs to be loaded into the main memory. The pointers in both algorithms, especially Twig^{2}Stack are complex and expensive to maintain.

To overcome these drawbacks, a new algorithm was proposed in [185] called TwigList. It replaced hierarchical stacks with lists (one for every query node) and pointers with simple intervals to capture structural relationships (i.e., a single recorded interval of contained elements for each child query). TwigList utilizes a stack to read all the document elements in pre-order and add them if they satisfy the mapping function conditions to the corresponding lists in post-order. For example, in Figure 3.10 element a_1 in $List_a$ has two intervals specified by four pointers in two 2-tuple, namely $< start_y, end_y >$ and $< start_x, end_x >$, $start_y$ records first element matches to a_1 as one of its descendants with y-tagged node while end_y records last element matches to a_1 as one of its descendants with y-tagged node. a_1 has $< 1,2 >$ as its recorded interval for contained elements corresponding to query node y.

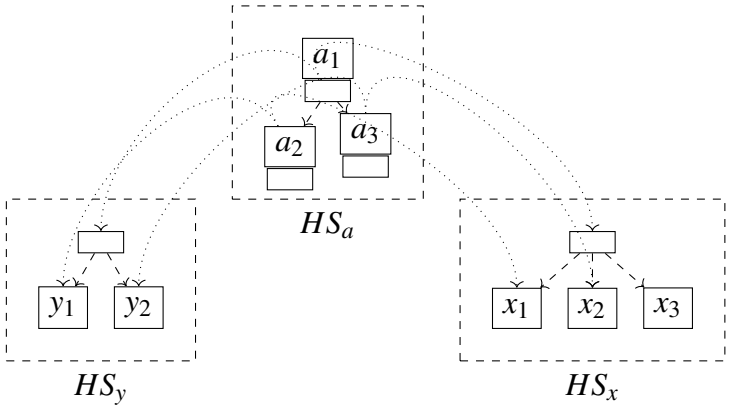
A comprehensive experimental study was conducted on both real-world datasets and synthetic datasets, and has shown that TwigList outperforms Twig^{2}Stack. The space and time complexity of TwigList are linear with respect to the size of the XML tree and the total number of elements in the output. A key to the efficient performance of TwigList is the storage of intermediate results in simple lists, but it assumes the relationship between elements in different lists is Ancestor-Descendant relationship. In order to handle Parent-Child edges TwigList utilizes extra pointers between elements in the same list indicating a sibling relationship. These additional pointers will degrade the result enumeration when twig queries contain Parent-Child edges. [132] extended TwigList by combining the features of two-phase holistic algorithms with the one-phase algorithms, namely TwigStack and TwigList. They incorporated into TwigList the filtering strategy applied in TwigStack to select useful elements before pushing them into the main stack in TwigList. They proposed two novel algorithms, called TwigMix and TwigFast, to improve the efficiency of TwigList. When twig pattern queries contain only Ancestor-Descendant edges both algorithms guarantee all elements in intermediate results contribute to the final results. According to their experiments TwigMix and TwigFast outperform TwigList.



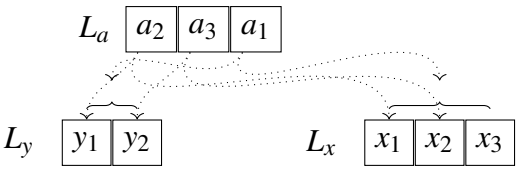
(a) an XML tree T_4



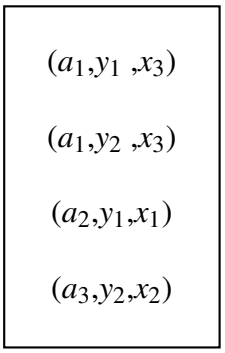
(b) a TPQ Q_1



(c) Twig^{2}Stack's trees of stacks



(d) TwigList's simple vectors



(e) matches

Figure 3.10: Illustration of Twig^{2}Stack and TwigList.

[89] assessed filtering strategies (a detailed analysis will be given in Chapter 4) and the linear time evaluation of the result enumeration in TwigList and its variant algorithm, TwigFast which proposed in [132] to examine document elements and store intermediate results in pre-order. The authors in [89] proposed a new storage scheme, level split approach which splits the intermediate list connected to its parent list with P-C edge to a number of levels equals to the depth of the XML tree as shown in Figure 3.11 for the data and query in Figure 3.10. In their paper, a combination of preorder and postorder filtering methods is adopted to develop two algorithms, namely: TJStrictPre and TJStrictPost. The experimental results have indicated the ability of the new method to eliminate useless elements in inner lists, and the size of intermediate results is by far smaller in comparison to TwigList and TwigFast. The new approaches can guarantee

linear CPU and I/O complexities of the output enumeration with respect to the output size. However, they suffer from large intermediate results in comparison with the query output. In [22], GTPStack improved the filtering strategy proposed in [89] by eliminating unnecessary self-nested matching checks (i.e., recursive calls) similar to that introduced in [130]. GTPStack is capable of processing GTPs efficiently.

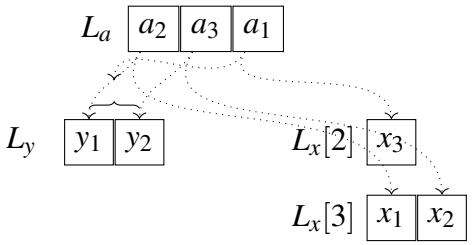


Figure 3.11: Illustration of level split list approach introduced in [89].

[70] extended twig pattern to make it more expressive to handle positional predicates and following-sibling relationship in XPath expressions. They developed a new algorithm called TwigPos which is a superset of TwigList for processing the new version of TPQs, or ExTwig for short. The work of [136] proposed an extension to TwigList in order to process recursive queries. The experimental results showed that the new approach is applicable and efficient. The main weakness with this algorithm is that it considers only one level of recursion among query nodes.

The algorithms described above of XML tree pattern concentrate only on tree pattern queries with A-D and P-C relations. Little research has been performed on tree pattern queries which may include order restriction, wildcards, and negation functions, all of them are repeatedly utilized in XML query languages such as XQuery and XPath. For instance, Figure 3.12 shows some examples of ordered twig pattern queries, the symbol " $<$ " is attached to branching nodes to indicate that its descendants are ordered (i.e., descendants must appear in the correct order according to the query). When a twig query includes a wildcard node represented by "*", which can match any single node. For example, consider the following a query *a//*/f*, the query asks for a f-tagged node that is a descendant of an a-tagged node but the f-tagged node has to be at least two level below its ancestor (i.e., a-tagged node). Usually wildcards are used to specify structural constraints on twig queries. To handle ordered twig patterns, previous work has to perform post-processing step to eliminate irrelevant results.

[145] was the first research paper to study the efficiency of evaluating ordered twig patterns in a holistic way without a post-processing phase. They introduced a new concept called ordered children extension (for short OCE) which expands the mapping function to satisfy one more condition that the order of sibling query nodes are also satisfied. Based on that OCE a novel holistic join algorithm was developed to evaluate ordered twig queries, called OrderedTJ which is an extension of TwigStackList. However, the authors did not take into account nesting elements within the XML data (more explanation on this issue will be given in Chapters 4 and 7).

According to [147] labelling scheme is commonly used to label an XML document to accelerate XML query performance by recording information on the path of an element to capture structural relationships rapidly during query processing without the need to access the XML document physically. Motivated by this, the authors proposed a novel labelling scheme based on existing Dewey labelling, called extended Dewey [211]. The proposed technique is a *mod* function to label an element based on its occurrence order appearance to its parent node among its sibling node(s). To elaborate, suppose there is a node that has three distinct child nodes, each child node will be assigned a local number such that the result of its reminder after dividing by 3 is either 0,1 or 2 according to the appearance of the tag name within the parent node. The purpose of this mechanism is to derive a node name from its label which leads to reduced I/O cost by only accessing elements of leaf nodes of a twig query since the proposed labelling scheme has the ability to turn a given node label into a series of elements names in the path from that node to the root.

The researchers extended the holistic twig pattern algorithm TwigStack proposed in [40] by applying their proposed labelling scheme instead of the region-based scheme used in TwigStack. The proposed algorithm was called TJFast and according to the experimental results shows a superior performance in terms of reducing disk access cost because only elements corresponding to leaf query nodes are accessed since "virtual streams" is used for internal nodes, by inferring the existence of elements from their descendants. The authors also considered a GTP query where optional axes and returned nodes are defined to present more semantics than a simple twig pattern query [53, 58]. As a consequence, they classified all nodes in GTP queries into four categories based on their properties and contribution to the final result. Thus, a new data structure was an augmentation of TJFast to maintain all descendants and children for generalized query nodes without buffering them into the main memory. The developed algorithm for evaluating generalized twig query is called GTJFast. Moreover, the authors exploited *tag* + *level* data partitioning scheme on the input lists to prune data by levels which speeds up the query processing evaluation when twig query contains more parent-child edges since the parent-child relationship is strictly specified by level. As a result, an extension was made to the previous proposed algorithms TJFast and GTJFast to apply *tag* + *level* partition technique and the proposed algorithms respectively are TJFastTL and GTJFastTL. Experiments were conducted to compare the proposed algorithms with TwigStack and TwigStackList. The proposed algorithms yield significant improvement in the query processing performance in terms of reducing disk access costs [147]. The authors in [118] augmented the original TJFast join algorithm with a set of children linked stacks to support the evaluation of ordered tree pattern queries, called OTJFast.

More recently, [146] examined a large series of XML-formatted tree patterns, referred to as extended XML tree patterns, which might contain negation, ancestor-descendant (A-D), parent-child (P-C) relationships, order restriction, and wildcards. The authors established a theoretical framework for optimal XML query processing. They concluded

(a) simple ordered TPQ (b) complex ordered TPQ

Figure 3.12: Illustration of ordered twig queries. "<" is used to identify ordered branching query nodes.

by identifying a new concept called matching cross, which describes a moment during query execution where the previous holistic algorithms have to decide whether to output redundant intermediate results or to miss potential results. The matching cross, thus, is the reason for the weakness of other comprehensive holistic algorithms. As a result, the researchers suggested a new algorithm to handle extended XML tree patterns effectively, called TreeMatch which is an extension to TJFast to avoid producing useless paths by returning target nodes. A group of empirical outcomes on both synthetic and real world datasets shows the efficiency and effectiveness of the proposed algorithm and theories. Unlike previous algorithms where the answers to queries are the whole matching solutions, TreeMatch only outputs elements corresponding to query nodes required to be returned in the query. The experiments reveal that the TreeMatch algorithm is optimal for queries with Ancestor-Descendant relationship in non-returned branching edges. It should be noted that when all query nodes must be returned, there is no difference between TreeMatch and TJFast.

Most of the holistic twig join algorithms following from TwigStack in [40] only handle plain twig patterns where each node corresponds to a particular element name and the siblings belonging to the same parent are connected via the AND logical operator, which is implicitly represented in the normal twig pattern. To illustrate, the following query, $a[/x]/y[/f\ and/d]$ selects a y-tagged node that must have two child nodes of types f-tagged and d-tagged, and has a sibling node x-tagged where both of the y-tagged and x-tagged nodes are children of an a-tagged node. The AND between the query nodes x and y is implicit, whereas the one between the query nodes f and d is obviously explicit. A twig query that has an arbitrary combination of Boolean operations is called a Boolean-twig or AND/OR/NOT twig (also referred to as B-twig). There are relatively few published papers in the area of XML query processing that have examined efficient evaluation of XML twig pattern queries with all or some of the Boolean logical operators [113, 249, 245, 106, 66, 68]. It should be noted that a sequence of predicates (i.e., []) can be used to replace conjunction of predicates (i.e., explicit AND operators) so that $a[/x]/y[/f\ and/d]$ would be reformulated as $a[/x]/y[/f][/d]$.

The naïve method to process XML twig queries with logical predicates is to decompose the twig query with OR or NOT to multiple twig queries, and then combine the final results

by computing the set difference (union) between them in case of NOT (OR) operator. The first holistic twig join algorithm to process twig queries with OR predicates was proposed in [113]. The key idea is to avoid the decomposition process which causes much more I/O and CPU overhead since some elements need to be accessed more than once in order to produce the result to a twig query with logical OR predicates. For example, the twig query $a[/x\ or/y]/f$ can be decomposed into two AND-twig queries $a[/x]/f$ and $a[/y]/f$, with the final result being generated by combining both results. Elements corresponding to a and f query nodes have to be accessed twice. Because of the decomposition mechanism equivalent to transforming an arbitrary logical expression into a logical expression in disjunctive normal form (DNF), the researchers introduced a new concept to evaluate logical OR predicates, called OR-block, which is a sub-tree attached to the original twig and the root is an OR node, the children of the OR-block sub-tree are query nodes connected to the parent of the OR-block sub-tree. The reason for this is to have twigs with logical AND operators whether explicit or implicit and OR-block(s). Every OR-block in their proposal is augmented with logical formula $P(n)$ to record the information needed when the OR predicate is evaluated. A new holistic algorithm was designed to process twig queries with OR predicates, called GTwigMerge.

[249] addressed the challenges for processing AND/NOT-twig queries holistically. They developed a new algorithm, TwigStackList$\neg$, which guarantees the intermediate results are always smaller than the decomposition-based method. The first attempt made to process a twig query with all logical predicates was in [245]. The authors proposed a new path-partitioned encoding scheme to derive all path information of elements in XML documents. A new simplified representation of a twig pattern was also introduced to represent the logical predicates in the ordinary twig pattern. Based on the their encoding scheme and the new representation of twig pattern, they proposed a new join algorithm to process twig queries without holistically computing the twig patterns. Nevertheless, developing a holistic twig algorithm that completely supports and is compatible with the three basic logical operators has been considered as challenging due to the fact that all the holistic twig join algorithms proposed in the literature and originated from TwigStack have failed to handle holistically a general B-twig query and provide good performance. One possible implication of this is that the main challenge as addressed in [68] is arbitrary occurrences of arbitrary combinations of AND/OR/NOT logical operators in B-twig patterns. They introduced a new mechanism to represent a twig query with an arbitrary combination of logical operations, which is derived from traditional Boolean normalization. The idea of this mechanism, called B-twig normalization, is to push all the NOTs symbols to the leaf query node and then applying the OR blocks technique mentioned earlier in [113]. Therefore, they proposed some supporting functions that can be added to any of the existing holistic twig join algorithms to make them able to handle complex B-twig queries after the normalization pre-processing step. In this paper, the authors proposed a novel holistic twig join algorithm, named BTwigMerge. An analytical and experimental study

was conducted and BTwigMerge, according to the results, showed a superior performance to related approaches. The same group in [66] examined the efficiency of processing Boolean twigs directly without normalizing the input B-twig query, because they claimed that the pre-normalization phase would incur extra processing time and in the worst case query expansion increases exponentially. They introduced a new mechanism, called status mechanism, in which every query node is associated with a boolean constant to indicate whether or not a match rooted at this query node is satisfied. The nodes in twig patterns are classified into two groups: query nodes and logical nodes. A query node is associated with an element tag-name in XML documents, while a logical node can be an AND, OR or NOT node. The researchers developed a new status updating mechanism based on the type of query nodes. An extension has made to TwigStack based on the status mechanism to process B-twig queries, the developed algorithm being called DBTwigMerge. According to the experimental results, the algorithm designed has failed to outperform the previous one with normalization in [68], BTwigMerge. In order to reduce the complexity of normalization, [67] introduced a new approach which combines the advantages of the previous ones. They used only three rules from the eight rules of BTwigMerge for transforming any B-twig into their desired form of B-twigs, called well-formed B-twigs (WFBT). According to the experimental results, the new approach, FBTwigMerge significantly outperformed the previous algorithms. On the whole,, once a TPQ with logical operators is normalized, it is straightforward to include boolean expressions in the filtering strategy [24, 22].

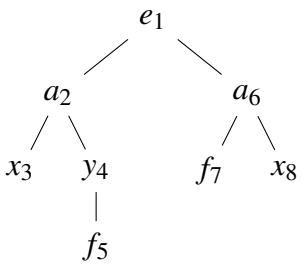
An XML query language such as XQuery and XPath allow the use of wildcards to broaden the scope of pattern matching, such a feature making the holistic evaluation of twig tree patterns with wildcards more complex. Therefore, optimizing XPath expressions by reducing the number of non-redundant wildcard steps is the work of [47]. An efficient rewriting technique was introduced based on a new composite axes, called layer axes which is a generalization of the other vertical axis, namely the Ancestor-Descendant and Parent-Child axes. They only proposed a rewriting algorithm and studied the effectiveness of the new proposed layer without implementation of the new algorithm to process the new axes. Recently, [237] proposed a novel approach to transform twig pattern queries with wildcards to equivalent wildcards-free tree patterns. They introduced a new axes, AD-dis, which is a generalize case of both Ancestor-Descendant and Parent-Child axes. By utilizing the new axes both branching and non-branching wildcards query nodes can be eliminated. The authors developed two algorithms namely, Path* and Twig*, to process path and twig queries, respectively. The most interesting aspect of their work is the rewriting optimization utilizing AD-dis (dis is an abbreviation for distance, providing relatively the same benefits as layer axes in [47]). The key problem with this approach is that they have to decompose a given twig pattern query into paths, then deploy rewriting paths utilizing AD-dis axis with recording the information of lowest common node(s) participating in different single rewriting paths. A serious weakness with this Twig* algorithm, however, is that single

paths are merged-joined based on the property of a prefix path labelling scheme, Dewey labelling, which incurs expensive iterations over the common ancestors to determine the merge-able nodes. A set of experiments were conducted to verify the effectiveness and the efficiency of the proposed algorithms, and the experimental results indicate that Twig* can guarantee ,in the worst case, I/O and CPU time complexities to be linear in the sum of the $(n-m)$ input lists and the output lists, where n is the number of query nodes and m is the number of * nodes.

More recently, new algorithms were proposed to speed up XML query processing by combining the well-known XML indexing types, namely structural summaries and node labelling schemes [105]. The TwigStack algorithm and its variants are based on node labelling schemes to capture the basic relationships between elements in XML documents in amortized constant time such as the region-based labelling scheme in [144, 40], the extended Dewey in [146, 147] and the Dewey in [19]. Most of the path indices mentioned in Section 3.2.1 were proposed to facilitate the naïve method (navigation-based approach) to evaluate tree-based data such as XML documents which requires traversing the entire document and performing many forward and backward traversals (especial when processing XML queries with A-D edges) to produce the answers to XML queries. In order to reduce the number of joins performed by TwigStack at the second phase, [96] proposed a new algorithm, TwigX-Guide, which is based on a DataGuide coupled with range-based labelling scheme. The key idea of this algorithm is that it combines the efficiency of the TwigStack algorithm, which is optimal in XML queries with A-D edges and the efficiency of a DataGuide which can process tree pattern queries with P-C edges efficiently in constant time. That results in a new algorithm that partitions XML tree nodes according to their occurrences in DataGuide single paths which leads later to reduce the disk-access costs by only reading elements that are likely participate in the final results rather than reading every element whose tag-name appears in the tree pattern query. Their technique in partitioning elements in XML documents is similar to prefix path labelling streaming proposed in [54]. Their algorithm needs to perform, in the worst case, one join operation on the top branching node while TwigStack and its variants need to merge-join the single paths produced in the first phase $n-1$ times where n is the number of query nodes. According to the experiments represented in [96] TwigX-Guide outperforms TwigStack and TwigStackList in terms of execution time for queries containing only P-C edges. When TPQs contain a mix of A-D and P-C, TwigX-Guide failed to outperform TwigStack and TwigStackList.

[105] proposed a novel method, called S^3 which can is a non-holistic approach which combines structural summaries and the inference of internal node matches similar to TJFast. The label lists contain DeweyIDs associated to the class of nodes in the structural summary "QueryGuide". The authors addressed the challenge in optimising XML query evaluation when processing large-scale XML documents which is to avoid document access as much as possible. A new mechanism was suggested to guide the query evaluation to reduce the search space by executing, first, tree pattern queries on the structural summary, called

QueryGuide which describes the structure of XML documents by its paths. Every path class in QueryGuide has a corresponding elements which are labelled using the Dewey labelling scheme and clustered together in lists in their ascending lexicographical order. Unlike the previous holistic joins, S^3 performs the tree matching on QueryGuide to select inverted lists that match path expressions over the structural summary. Only elements whose tags appear as leaves in tree pattern queries are joined based on the property of the Dewey labelling scheme to generate the query answers. Figure 3.13 depicts the QueryGuide and its corresponding inverted lists. For example, consider the query $TPQ = //a[/x]/y$. The previous algorithm i.e, TwigStack will retrieve all elements whose tags appear in the query *a,x* and *y* in this example, but it can be seen from the structural summary that only elements corresponding to query nodes *a* in class a_2, *x* in class x_3 and *y* in class y_4 satisfy the query matching conditions. Scanning, thus, *a* and *x* in classes a_6 and x_8 will be redundant and cause extra I/O overhead. The researchers indicated that any previous tree matching algorithm can be performed first on the structural summary to retrieve the participant node classes since the size of the QueryGuide is much smaller than the source data tree. S^3's evaluation mechanism is similar to TJFast proposed in [147]. That is, S^3 looks at the query leaf nodes pairwise and merge joins sets based on their lowest common ancestor query nodes using the property of Dewey labelling scheme. Thus, large sets of unused intermediate results may be produced. In [105], S^3 was compared with algorithms (i.e., TwigSack, TJFast and TwigList), which do not use structural summaries, in terms of memory consumption, response time and disk-access costs. A better study would compare the performance of S^3 with TJDewey [19] or the approach proposed in [102], CIS-X which extended TwigList to use structural summaries.



(a) QueryGuide

e_1	1
a_2	1.1:1.9:1.23
x_3	1.9.2:1.23.2
y_4	1.1.2:1.1.4:1.9.1
f_5	1.1.2.1
a_6	1.2:1.3:1.4
f_7	1.2.1:1.4.1
x_8	1.2.2

(b) label lists

Figure 3.13: (a) QueryGuide as structural summary and (b) its corresponding label lists.

A twig pattern matching algorithm, called TwigVersion was proposed in [240]. The authors proposed a new labelling scheme, called version-labelling scheme to identify the repetitive structures (twigs) in XML documents. The proposed labelling scheme was incorporated into a structural index tree to form a version tree (DataGuide annotated with Dewey labelling scheme) where each node in it can be identified by its unique label

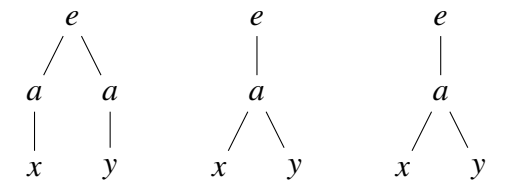
path. The TwigVersion algorithm executes tree pattern queries in bottom-up fashion by inspecting version information associated with every node in the version tree (i.e., F&B summary). The authors also proposed a new technique to compress content information inside the version tree to support tree pattern queries with selection value-based predicates. The experimental results revealed that TwigVersion significantly outperforms TwigStack, TJFast and Twig^{2}Stack algorithms and its performance improvement has a strong dependency on the size of version tree and the ratio of the number of versions to the total element in XML documents. [137] proposed a new algorithm to evaluate XML tree pattern queries in three phases, where the first phase is to perform TwigStack on version tree similar to the one proposed in [240] to produce the structural matching, after which TJFast is executed on the retrieved version nodes to produce the final query answers. The experimental results showed that the performance of the algorithm, Twig3Version, may be degraded when the structure of XML documents is very complex.

Summary Finding all occurrences of tree pattern queries (TPQs) in XML documents is considered as a specific task for XML query processing of XML databases. The groundbreaking work of [40] established for the first time that processing XML tree patterns can be performed holistically without decomposing tree pattern queries into multiple binary joins. The key idea to their work is to merge multiple inverted lists at a time in pipeline strategy to reduce the size of intermediate results. Two algorithms were proposed, namely PathStack to process simple path queries holistically and TwigStack to answer twig queries in a holistic way by decomposing twig queries into multiple root-to-leaf paths. TwigStack uses a linked of stacks chain to represent intermediate results and works in two phases. The first phase is to output single path solutions and the second phase is to merge all path results produced in the first phase to produce the answer to the whole twig query. As been discussed above in this section, several approaches have been published in the literature to improve the efficiency of the TwigStack algorithm since it only optimal for tree pattern queries having only Ancestor-Descendant edges. The researchers in the field of XML query processing community have extended TwigStack in order to enlarge the class of queries for which the method is optimal (no redundant paths generated in the first phase) as in [144, 54, 131], support selective queries through value predicates [236], reduce I/O costs and avoid document access as much as possible [147], eliminate merge join operations in the second phase [185, 53, 132, 89, 22] and enhance execution time by utilizing a combination of different XML indexing techniques [95, 19, 105, 96]. Little work has been suggested to evaluate logical predicates in XML tree pattern queries holistically such as [113, 249, 66, 68, 22, 67]. Skipping useless elements techniques such as XB-tree, XR-tree and virtual cursors can be easily incorporated into algorithms for structural joins with few extensions. Finally, searching structural summaries (e.g., DataGuide) is time consuming [21], therefore efficient query pattern search of large structural index trees is needed. To sum up, holistic approaches may be classified on the basis of the output enumeration algorithm into top-down (two-phase) algorithms which use the output

enumeration process introduced in TwigStack and bottom-up (one-phase) methods which use the output enumeration algorithm introduced in Twig^{2}Stack and TwigList.

3.3.2.3 Sequence-Based Approaches

The result of an XML query is a set of nodes that is sorted in document order which causes generation of duplicates that need to be eliminated before producing the final results. The key issue of efficient evaluation of XMl queries is to avoid duplicates at any time during processing. A new approach was proposed to evaluate XML queries to avoid merge-join operations by transforming both XML documents and queries into sequences [230, 188, 213]. The process of XML query processing, then is reduced to subsequence matching. Virtual Suffix Tree, or ViST in [230] transformed XML documents and queries into structure-encoded sequences. It builds virtual tries (two levels of B+-tree) so that sub-sequence matching is performed through a series of index probes. The answers to queries generated by ViST may result in some false positives. To tackle this limitation, [188] proposed an indexing technique to transform both the XML documents and queries into sequences by the *Prüfer* code method that generates an one-to-one correspondence between trees and sequences. This, thus, avoids the false answers problem generated by transforming trees into sequences. The new algorithm is called PRIX and has to perform expensive post-processing refining phases to eliminate irrelevant results (i.e., possible matches) before producing the final answers to queries. Figure 3.14 illustrates the false positives problem produced by previous sequence-based algorithms. To evaluate the query Q_1 over the data T_1, it can be noticed that the underlined sequences form an answer to Q_1. However, it is incorrect because element x and element y in the XML tree T_1 do not have the same parent. The limitations of sequence-based approach can be summarised into two main drawbacks. Firstly, it supports ordered twig queries only because it is a sub-sequence matching process rather than tree matching process, and secondly, it might unnecessarily scan some nodes more than once.



(a) an XML tree T_1

(b) an XML tree T_2

(c) a TPQ Q_1

T_1	(e,ε) (a,e) (x,ea) (a,e) (y,ea)
T_2	(e,ε)(a,e)(x,ea)y,ea)
Q_1	(e,ε)(a,e)(x,a)(y,a)

(d) underlined indicates subsequence matching

Figure 3.14: Illustration of false positive problem in ViST [188].

This section has reviewed the process of evaluating XML queries as sub-sequence matching. The most sophisticated algorithms in this approach were discussed. The lack of algorithms in this approach may be due to the fact that the number of sequence indices is

limited and they are built upon these indexing methods to transform XML documents and queries into sequences.

3.4 Conclusion

This chapter has reviewed the major approaches for an XML query processing. The core operation for processing XML query is examining and searching for the outcrop of a twig pattern query in an XML database. As mentioned in this chapter, many algorithms have been proposed in the literature. The structural join approaches are the oldest and derived from the traditional join approach exploited in relational databases by providing a native implementation for processing XML queries. The main issue with algorithms in the structural join approach is that they may create large valueless intermediate results because of the results of the individual binary relationships may not contribute to the final result. The holistic twig join algorithms were proposed to solve this problem, and the first holistic twig join algorithm is TwigStack. Subsequently, various holistic twig join algorithms were proposed to improve the original one in terms of reducing the disk access cost as in [54, 147, 125], avoiding expensive merging phase as in [185, 89, 22] or evaluating some or all logical predicates (B-twigs) [68, 66]. Moreover, non-holistic approaches were proposed to perform twig pattern matching originated from TwigStack on indexing tree to enhance query performance and minimize I/O cost [240, 105, 96]. Additionally, sequence matching algorithms show significant improvement over the structural join algorithms yet they are limited due to the fact that they are reliant on indexing methods to transform the XML documents into sequences and they only support ordered twig pattern queries and it is not trivial task to modify them in order to support unordered twig pattern queries.

To conclude, the purpose of this chapter was to review the literature on XML query processing. It begins by introducing the main concepts and techniques exploited in XML retrieval systems. Then, the process of query evaluation in XML query processing is viewed as a tree matching process and it is classified into approximate and exact matching. This chapter, also, has reviewed the three key aspects of the exact tree matching process for XML retrieval systems which are structural joins, holistic joins and sequence-based approaches.

In view of the success and limitations of previous work, this thesis aims at studying the space-time tradeoff for holistic twig matching algorithms and focuses on optimizing space consumption without sacrificing query response time. The next chapter will discuss problems arising from existing approaches in order to identify a place where a new contribution could be made. Furthermore, the research hypothesis will be highlighted.

Chapter 4

Research Hypothesis and Methodology

4.1 Introduction

This chapter will outline the research motivation and describe the general area of research in order to address the potential research problems emerging from the literature review in the field of XML query processing (see Chapter 3). The main interest of this research revolves around the matching filtering concepts introduced in [89], and how it can be realized efficiently by extending the existing labelling schemes to overcome the issues arising when developing an XML query model as a part of *native* XML databases management systems (XDBMS) [9, 85, 86]. In *native* XML databases management systems, a comprehensive set of core functionalities designed for data stored in XML format is offered. However, this thesis concentrates on the basic functionality of the query model in these systems and describes the research work in a particular area of XML matching process. In the context of XML query processing, Twig Pattern Query (for short TPQ) is the simple query model proposed in the literature [20, 22, 118]. A twig pattern is a small tree and can be represented as a rooted, labelled tree [22, 5, 40, 38, 70, 145, 237, 133, 128, 236, 207].

The purpose of this chapter is to discuss the problems identified by the literature survey conducted in the literature review chapter (see Chapter 3). This chapter then demonstrates the main methodology and research techniques which have been used in the field and will be adopted in this thesis to establish the overall research process used to test the tentative hypothesis presented later [210, 179, 193, 69, 179].

The rest of this chapter is organized as follows. The research problems and motivation will be explained in Section 4.2. The research hypothesis will be presented in Section 4.3 as will the methodology used in this research is described. The scope of this thesis is highlighted in Section 4.4 and Section 4.5 will demonstrate the main objectives of the solution proposed.

4.2 Research Problems and Motivation

The growing number of XML documents leads to the need for appropriate XML querying algorithms which are able to retrieve data efficiently within XML documents [6, 110, 31, 98, 209]. In recent years, there has been an increasing interest in twig pattern matching [132, 5, 20, 22, 40, 18, 38, 43, 54, 53, 7, 89, 148, 24]. Due to the importance of TPQs in XML retrieval systems, finding all occurrences of a tree pattern query in an XML document is often considered as a specific task for XML databases as well as a core operation in XML query processing. However, it could be argued that most of the existing algorithms fail to process XML twig pattern queries efficiently or to guarantee an optimal evaluation without storing useless elements in intermediate storage prior to forming the final answer regardless of the query class as was mentioned in Chapter 3.

Most existing XML query processing algorithms [54, 20, 237, 207, 144, 7, 132, 118, 116, 22, 89] rely on XML indexing techniques to scan only XML data relevant to XML queries, as a result, the XML query performance is improved. In XML, there are two basic type of indices. The first one is to index each node in an XML document for recording its positional information [253, 90, 81, 98, 211, 147, 238, 175, 186, 184]. This group is well-known as node label or labelling schemes. In this group of indices, every node in an XML document is assigned an unique label to record its position within the original XML tree. The labelling scheme should enable determination of the structural information, namely: child (also is referred to as Parent-Child) and descendant (short for Ancestor-Descendant) relationships. Therefore, for any given two elements in an XML document, the relationship between them (if it exists) can be computed in constant time.

The alternative uses root-to-node paths in the XML document and is well-known as graph indexing (also referred to as structural summary or path indexing). Because an XML document can be modelled as rooted, ordered, labelled tree, a labelled path is defined as a sequence of tag names in the form of $tag_1/tag_2/\ldots/tag_n$ from the root represented by tag_1 to node n tagged by tag_n . For illustration, consider the XML tree in Figure 4.1a, elements tagged by *a* can be stored in different storage structures according to their unique labelled paths. Consequently, elements corresponding to the path e/a are $\{\ a_1\ \ a_3\ \ a_4\ \}$, while a_2 is stored alone in its distinct labelled path $e/a/a$. A classic example of a path index is the *DataGuide* which is a typical model that merges all equivalent paths into a single one. Every path in *DataGuide* is unique. One major drawback of this approach is that it only supports a simple path queries, however, there are some graph indices that cover twig path queries as in [119] but one of the limitations with these indices is that they can be very large. An example of a *DataGuide* for the same XML tree is depicted in Figure 4.1b. The literature review chapters (see Chapters 3) highlighted the alternative indexing techniques of XML proposed in the literature and studied their advantages and limitations in depth [174, 81, 9, 76, 244, 253, 240, 52, 119], but these two are important for a wide range of XML query processing algorithms [20].

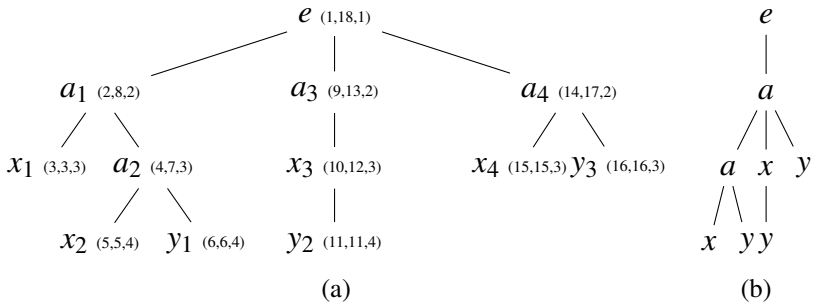


Figure 4.1: (a) sample of an XML tree and (b) its *DataGuide*

Both node and graph indexing are essential to XML query processing algorithms, they play important role in providing efficient evaluation of queries with respect to time complexity and memory consumption overhead [20, 188, 240, 6]. For the sake of simplicity the following example 4.1 aims to explain the use of labels in determination of hierarchical relationships in XML trees, and Example 4.4 is a demonstration of path indexing technique used in facilitating XML query processing.

Example 4.1. *Consider the Figure 4.1a, the structural relationships between the elements can be determined according to property 4.3 and 4.2 for ancestor-descendant and parent-child relationships, respectively. Consider the relationship between node* a_1 *and* y_1*, as the elements are labelled based on containment labelling scheme proposed in [253].* a_1 *is an ancestor of* y_1 *because* $2 < 6 < 8$*. Also,* a_1 *is a parent node of* a_2 *because the parent-child conditions are satisfied as* $2 < 4 < 8$ *and* $2 + 1 = 3$*.*

Property 4.2 (Parent-Child relationship)**.** *Element u is a parent of element v if and only if*

$$start_u < start_v < end_u \text{ and } level_u + 1 = level_v$$

Property 4.3 (Ancestor-Descendant relationship)**.** *Element u is an ancestor of element v if and only if*

$$start_u < start_v < end_u$$

Example 4.4. *Suppose a simple path query* $Q = /e/a$ *is issued against the XML tree in Figure 4.1a and by traversing the DataGuide in Figure 4.1b, the result of this query can be returned by a single access because it is a simple path so that elements corresponding to this labelled path in a DataGuide can be retrieved in a single access. Only* a_1*,* a_3 *and* a_4 *are the answer to Q and* a_2 *is discarded.*

This section aims to provide a general background to different techniques used as sub-components in XML query systems because it is necessary here to clarify exactly what is meant by deploying XML indexing in the context of XML query processing before proceeding to criticize the previous approaches. Chapter 2 has highlighted the different types of XML queries and their characteristics in more details. As a result, the research

$$
\begin{array}{l}
TPQ \rightarrow Pattern \\
Pattern \rightarrow Step(Predicate)^{*}\ (Pattern)? \\
Step \rightarrow Axis\ NodeTest \\
Predicate \rightarrow [\ Pattern\ ] \\
Axis \rightarrow\ /\ |\ // \\
NodeTest \rightarrow String
\end{array}
$$

Figure 4.2: A grammar of TPQ

presented in this thesis is motivated by the need to address a set of major problems derived from the literature survey conducted in Chapters 3. The research problems are summarised in the next sections.

4.2.1 Determination of the Basic Structural Axes

An Extensible Markup Language (XML) tree pattern query is defined as a complex selection on elements of an XML document specified by structural information of the selected elements [132]. Improving the efficiency of tree patterns matching is a core operation in processing of XML queries. Therefore, XML query processing has attracted a considerable number of researchers to improve its performance [110, 147, 144, 249, 146, 15, 53]. One of the most important problems in XML query processing is tree pattern matching. Generally, tree pattern matching is defined as mapping function M between a given tree pattern query Q and XML data D, $M: Q \rightarrow D$ that maps nodes of Q into nodes of D where structural relationships are preserved and the predicates of Q are satisfied. Formally, tree pattern matching must find all matches of a given tree pattern query Q on an XML document D [81, 209, 40, 5, 68, 146]. The matching problem, then can be formalised in Definition 4.10. In addition, *weak* and *strict* filter matching are defined in definitions 4.15 and 4.16, respectively adopted from [89]. As has been discussed in the literature review chapter (see Chapter 3), most of the previous holistic twig pattern processing approaches are reliant on a combination of matching and filtering techniques in order to process twig queries efficiently. *subtree* and *prefix-path* are two of the most widely used groups of matching and filtering strategies in the literature. For illustration, the definitions of both are given in the Definitions 4.14 and 4.13, respectively. The grammar of TPQ used in this thesis which is expressed as a fragment of the grammar of *XPath 2.0* [222] is introduced in Figure 4.2.

In XML, both data and queries are represented and expressed, respectively, using tree-structured model, the following are the definitions of XML tree and twig pattern tree [209].

Definition 4.5 (XML Tree)**.** *A rooted, node-labelled tree is defined as* $T = (V, E, r, \sum_V, \mu)$ *where*

- $V = \{v_1, \ldots, v_n\}$ *is a finite set of nodes.*
- $E = \{(u, v) \in V \times V\}$ *is a set of edges.*

- $r \in V$ *is a distinguished node called the root.*
- $\sum_V$ *is the set of element names appearing in T.*
- $\mu : V - \{r\} \rightarrow \sum_V$ *is a labelling function which associates an element name with each node other than the root.*

The level of any node in T is the number of distinct element(s) along the unique path from itself to the root, where level(r) = 1.

Definition 4.6 (Child Relationship)**.** *Given two nodes u and v in a rooted, labelled tree where* $u, v \in V$*, v is a child of u if and only if* $\exists e \in E : e = (u, v)$*. Conversely, u is a parent of v, this relationship is denoted as PC or P-C.*

Definition 4.7 (Descendant Relationship)**.** *Given two nodes u and v in a rooted, labelled tree where* $u, v \in V$*, v is a descendant of u if and only if* $\exists n_1, \ldots, n_k \in V$ *such that* $(u, n_1) \in E$, $(n_1, n_2) \in E, \ldots, (n_k, v) \in E$ *where* $1 \leq k <$ *the depth of the tree. Conversely, u is an ancestor of v, this relationship is denoted as AD or A-D.*

The difference between an XML tree and a twig pattern can be seen in their types of edges, the XML tree is only entitled to have parent-child edges connected its nodes, while the twig pattern is extensible to handle the Ancestor-Descendant structural relationship as edges connected to its nodes. In practice, twig pattern is much more smaller than the original XML tree. Twig pattern can be seen as the translation of user query and translating an XML query plan into a twig pattern is not a simple task [93]. Complex XML queries are divided into several twig patterns because a single twig pattern can represent only a single XPath path expression. The complexity of XML queries determine the difficulty of translating them into twig pattern(s) [161]. In XML query optimization, the process of translating user queries to twig patterns, then optimise the generated twig patterns has been considered as the most effective solution used in the literature [93].

Definition 4.8 (Twig Pattern)**.** *TP is a rooted, node-labelled tree* $TP = (V, E, r, \sum_V, \mu)$ *where*

- $V = \{v_1, \ldots, v_n\}$ *is a finite set of query nodes.*
- $E = \{(u, v) \in V \times V\}$ *is a set of edges which represents parent-child or ancestor-descendant relationships between connected query nodes. The set of child edges is denoted by* $E_{/}$*, while the set of descendant edges is denoted by* $E_{//}$*.*
- $r \in V$ *is a distinguished query node called the root.*
- $\sum_V$ *is the set of element names appearing in TP.*
- $\mu : V \rightarrow \sum_V$ *is a labelling function which associates an element name with each node.*

Definition 4.9 (Twig Pattern Query)**.** *TPQ is a pair (TP,F) where*

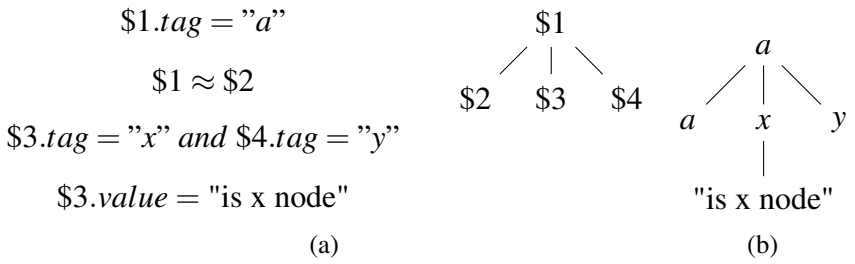


Figure 4.3: (a) sample of a twig pattern with formula F and (b) its optimised transformation to more readable one.

- *TP is a twig pattern.*
- *F is a formula which specifies constraints on TP's nodes.*

[122] explained how the formula F is expressed in a twig pattern. In their work, F is defined as a combination of tag constraints (TCs), value based constraints (VBCs) and node identity constraints (NICs). TCs specify constraints on tags (labels) of twig pattern nodes, e.g. node.tag(node.label) = "a" (see Figure 4.3). VBCs include selection constraints on values using relational operators $=, \neq, \geq, >, \leq$ *and* $<$. For illustration, consider the query a[/a][/x="is x node"]/y and its twig pattern representation in Figure 4.3, VBCs indicate that the x-node to be selected is associated with a text value equals to "is x node". NICs determine if two nodes of TP are the same using " $\approx$ " [93]. Consequently, the following definition formalises answers to twig pattern queries using matchings.

Definition 4.10 (Query Matching)**.** *A match of a twig pattern query* $TPQ = (TP = (V_1, E_1, r_1, \sum_{V_1}, \mu_1), F)$ *in* $D = (V_2, E_2, r_2, \sum_{V_2}, \mu_2)$ *is a total mapping* $M : TP \rightarrow D$ *that maps nodes of TP into nodes of D such that:*

- *For each query node* $u_1 \in V_1$ *maps to* $u_2 \in V_2$ *, then* $\mu_1(u_1) = \mu_2(u_2)$.
- *For each edge* $e = (u, v) \in TP$ *where e represents parent-child relationship, M(v) is a child of M(u) in D.*
- *For each edge* $e = (u, v) \in TP$ *where e represents ancestor-descendant relationship, M(v) is a descendant of M(u) in D.*
- *formula F of TPQ is satisfied.*

It is worth noting that the matching result of a twig pattern query as in Definition 4.11 coincides with that presented in the previous holistic twig pattern query processing approaches [81, 209, 40, 5, 68, 146, 108]. In contrast to this, *XPath* outputs only the last step of the path expression which is not part of a predicate. On the other hand, *XQuery* can define the output more precisely. In their examination of incorporating TPQ into the *XQuery* algebra [161] concluded that their developed algebra uses TPQ has a single output query node. Even when a complex *FLWOR XQuery expression* indicates more than one query node as the output of a TPQ, the twig is bound by a single query node in each

iterative of **for** clause [18]. Interestingly, it is almost certain that reducing the number of output query nodes in the final matching tuples has a significant impact on the performance of holistic TPQ processing approaches [146]. In this thesis, however, the output of TPQ is consistent with the previous holistic TPQ processing approaches as defined in Definition 4.11 in order to compare the proposed approaches with the state of the art approaches based on the processing time and memory consumption overheads.

Definition 4.11 (Matching Result Set)**.** *The matching of the query nodes of Q under a mapping of Q to D is a solution of Q on D. The answer of Q on D is an ordered set of all the solutions of Q on D. The answer to TPQ Q with n nodes can be represented as an n-ary relation where each tuple $(q_1, \ldots, q_n)$ consists of the database elements that identify a distinct match of Q in D. Some of the fields may be duplicated and some may not be in the document order, but tuples have to be in a sorted order of the common prefixes of the individual root-to-leaf paths.*

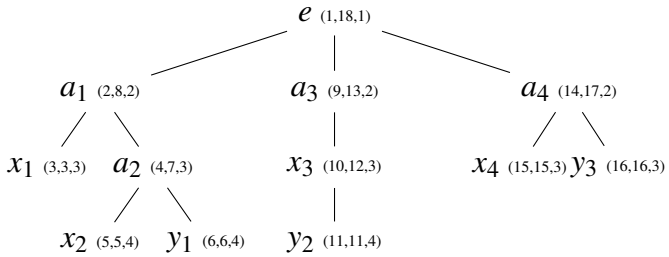


Figure 4.4: An XML tree.

Example 4.12. *Consider the XML tree in Figure 4.4, one possible solution of the twig pattern query Q = a[/x]/y is the tuple consists of (a_1, x_2, y_1), but the full answer to the query is the ordered set of all the matching solutions as (a_1, x_2, y_1) and (a_4, x_4, y_3).*

Definition 4.13 (Prefix-Path Matching)**.** *A query node in a TPQ Q as $q_n \in Q$ is a strictly/weakly prefix-path matching of an XML element $M(q_n) \in D$ if and only if the simple path $q_1, \ldots, q_n$ is a strictly/weakly match of q_n, where q_1 is the root query node.*

Definition 4.14 (Subtree Matching)**.** *A node in a TPQ Q as $q_n \in Q$ is a strictly/weakly subtree matching of an XML element $M(q_n) \in D$ if and only if all query nodes which are child or descendant nodes of q_n are in a strictly/weakly prefix-path matching of the simple paths starting from q_n as the root to each one of its children and descendants in Q.*

In reviewing the literature, the main research problem in the context of XML query processing models may be seen as

> "Given a twig pattern query Q and an XML data tree D, find all occurrences of Q in D efficiently.'

In the literature review chapter (see Chapter 3), the state-of-the-art of XML query processing algorithms were discussed. The evidence from this study suggests that most of the existing algorithms, regardless the query class, fail to process XML twig pattern queries efficiently or to guarantee an optimal evaluation without storing useless results in intermediate storage before computing the final answers. The classical holistic twig join algorithm *TwigStack* considers only the ancestor-descendant relationship between query nodes to process a twig query efficiently without storing irrelevant nodes in the intermediate storage. It has been reported [40, 61] that it has the worst-case I/O and CPU complexities when all edges in twigs are "//" (AD relationship) linear in the sum of the size of |Q| input and output lists where |Q| denotes the number of nodes in Q. However, it can not efficiently process twigs with "/" (PC relationship) edges or a combination of Ancestor-Descendant and Parent-Child relationships, as was explained in Chapter 3. Another limitation of *TwigStack* [40] is that it might perform unnecessary processing by considering weak *subtree* filtering between query nodes and strict matching between query nodes in each root-to-leaf path [89]. The weak filtering is performed in the core function *getNext()* in which an element e_n is considered likely to contribute to the result if and only if e_n has a descendant extension that is (i) e_n has a descendant element e_{n_i} in each of the streams corresponding to its child elements where e_{n_i} = children(e_n) ; (ii) each of its child elements satisfies recursively the first property (i). The latter is achieved by maintaining a chain of stacks corresponding to query nodes and each stack points to the stack corresponding to its parent query node and the root query node points to *null* to indicate the end of path. The strict path matching check is implied by a modification to the *push* method where elements are pushed onto its corresponding stack if and only if they have a *strict match* with the top elements in their parent stack as in the Definition 4.16. Note that the root query node satisfies the strict match since it is the ancestor of all query nodes.

In their comprehensive analysis of filtering strategies [89], the researchers have drawn a conclusion that the efficiency of XML matching algorithms could be improved by combining different types of filtering strategies. As a result, filtering strategy is a dominant feature of the efficiency of XML query matching algorithms. The use of *semi-strict* filtering has not been investigated and one possible definition, which is close to that of [89], has been broadened in this research to include inferred information from labelling schemes or structural indices, further explanation of this information will be found in Chapter 6 . This definition can be formalized as in Definition 4.17.

Definition 4.15 (Weak Matching)**.** *A node map according to M weakly satisfies the edge* $e = (p,q) \in Q$ *if M(p) is an ancestor of M(q).*

Definition 4.16 (Strict Matching)**.** *A node map according to M strictly satisfies the edge* $e = (p,q) \in Q$ *if and only if M(p) has a relationship with M(q) as specified by label(e).*

Definition 4.17 (Semi-Strict Matching)**.** *A node map according to M semi-strictly satisfies the edge $e = (p,q) \in Q$ if and only if M(p) is a ancestor of M(q) and it can be derived that M(p) has a relationship with M(q) as specified by label(e).*

TwigStack has been extended extensively in the literature [1], strategies to enhance it might involve reducing the number of partial paths when parent-child axes "/" exist in twig queries as in [54, 144, 18], alleviating the I/O cost by maintaining virtual cursors to avoid reading elements corresponding to internal query nodes [147, 125, 146], boosting processing time by skipping irrelevant elements in the input inverted lists based on prebuilt indices (variants of B-tree index) such as [40, 111], or avoiding redundant computation by sequentially forwarding the cursors to potential, matching elements [255, 133, 111, 96].

Nevertheless, [61] points out that the *TwigStack* algorithm and its variants which depend on a single sequentially scan of the input lists can not be optimal for evaluation of tree pattern queries with any arbitrary combination of ancestor-descendant and parent-child relationships. In the context of XML twig processing, a twig matching algorithm is considered optimal if and only if all elements stored in main memory are directly relevant to the twig solutions. It means that storing elements using simple, as in [144], or complex, as in [131], buffering techniques does not guarantee optimal evaluation since extra computations would cause overheads and violate the worst-case space complexity in the early phases prior to producing final solutions to twig queries. Despite this, reducing the irrelevant elements stored in main memory would improve the overall performance significantly as been reported in the literature and guarantee the linear worst-case complexity [53, 125, 144, 89, 54, 88, 22].

Numerous studies have attempted to devise an optimal evaluation for twig pattern queries over the past decades [53, 255, 105, 21, 128, 96, 66, 139, 70, 116, 240, 185]. The most obvious finding to emerge from the analysis (see Chapter 3) is that their main limitations, however, lie in the fact that they store many irrelevant nodes in intermediate storage and no optimal approach has been proven yet [20, 22, 89, 116, 132]. It is worth noting that the term *optimal* is misleading because the existing research recognizes an optimal evaluation of twig queries based on the approach chosen. If twig queries are processed in one of the two-phase group of algorithms, the optimal evaluation is achieved if and only if all partial single paths are part of the final results, while the one-phased family of algorithms achieve optimal processing if and only if all elements stored in intermediate storage are relevant to twig queries answers [22]. However, an optimal enumeration can be obtained in one-phase algorithms through the use of split vectors or tree of stacks as the intermediate storage used where useless elements corresponding to internal query nodes are pruned efficiently before performing the enumeration [53, 89]. In other words, the optimal evaluation guarantees the optimal enumeration but the opposite is not true. It has been pointed out that state of the art algorithms *TJStrictPre*, *TJStrictPost* and *GTPStack* still have redundant computation and overheads in practice, and store useless elements

[1] A more comprehensive study can be found in the literature review discussed in Chapter 3.

corresponding to leaf query nodes in the intermediate storage [22, 20, 89]. Furthermore, holistic twig algorithms define query classes for which they are optimal. A summary of the cutting edge XML query processing algorithms and their optimal set of query classes is presented in Table 4.1. Notice two important assumptions ensuring the linear I/O complexity are that a holistic algorithm can access only a single node from each stream in each step during the processing time and cursors can be only forwarded. It is believed that TPQ algorithms's optimal processing is based on the labelling schemes and *tag stream* schemes used and the characteristics of XML documents [22].

Table 4.1: A summary of previous algorithms and their filtering properties.

Algorithm	Source	Filtering		Intermediate Results		Optimal	Query Types
		preorder	postorder	paths	element references		
TwigStack	[40]	✓		✓			A-D
TwigStackList	[144]	✓		✓			P-C below non branching
iTwigJoin+TL	[54]	✓		✓			P-C
iTwigJoin+PPL	[54]	✓		✓			one branching query node or all P-C before A-D
$Twig^2Stack$	[53]		✓		✓	✓	optimal enumeration
$\frac{1}{2}PathStack + Twig^2Stack$	[53]	✓	✓		✓	✓	optimal enumeration
$\frac{1}{2}TwigStack + Twig^2Stack$	[21]	✓	✓		✓	✓	optimal enumeration
TJFast	[147]	✓		✓			P-C below non branching
TJFast+TL	[147]	✓		✓			P-C or P-C below non branching
TwigStackSorting+PPL	[18]	✓		✓			same as *iTwigJoin+PPL*
TreeMatch	[146]	✓					AD in non output branching edges
TwigList	[185]		✓		✓		
TwigFast	[132]	✓			✓		A-D
TJStrictPost	[89]	✓	✓		✓		A-D
TJStrictPost+ Split Vector	[89]	✓	✓		✓	✓	optimal enumeration
TJStrictPre	[89]	✓			✓		A-D
TJStrictPre+ Split Vector	[89]	✓	✓		✓	✓	optimal enumeration
GTPStack+ Split Vector	[22]	✓	✓		✓	✓	optimal enumeration

The work of [9] addressed the limitations of information encoded within labels produced by existing labelling schemes. It focused on performing join operations more earlier at leaf levels where the selectivity of query nodes is at its peak for data-oriented XML documents. The significance of the proposed approach stems from a comprehensive labelling scheme that could infer additional structural information, called *Nearest Common Ancestor, NCA for short* rather than the basic relationships among elements of XML documents. A particular problem in TPQ is *wildcard*. *A Twig pattern query* may contain a query node where its node label can be any node label in the XML documents, it is represented in Q as "*" (also referred to as, *wildcard* node). The authors in [237], proposed a new XML matching algorithm to process *wildcard* nodes efficiently by transforming twig pattern queries with one or more *wildcard* nodes into wildcard node free queries. Their success was made through devising a new rewriting mechanism and augmenting the elements' labels, called *Gap Label* with extra information to accelerate the matching process. In addition, this suggests a need to consider the amount of information stored within elements of XML documents to enhance the efficiency of XML query algorithms.

Example 4.18. *Consider the XML tree in Figure 4.1a, assume a simple path query to return only elements labelled y which have an element labelled a as a grandfather of it regardless of the label of its parent. In other words, a is exactly a parent of its parent. Such a query can be written as* $Q = a/*/y$*, then the query solutions consist of two tuples* (a_1, a_2, y_1) *and* (a_2, x_1, y_2).

To overcome the above issues, this thesis proposes a new approach that combines techniques from different aspects of XML query processing. The new approach is introduced in Chapter 6 where its features are related to this research problem. The key idea of is to find an appropriate, refined labelling scheme such that, for any given query node in Q, the set of its child query nodes in the XML document D specified by the major bottleneck structural relationship, namely Parent-Child can be determined efficiently. The proposed approach should be able to combine multiple filtering strategies to eliminate eventually storing useless elements in intermediate storage, by increasing the overall performance while space overheads are reduced. The next section is to address *ordered twig pattern query, or for short OTPQ* in which the document order of XML elements is taken into account during the matching process [118, 145, 141, 207, 218].

4.2.2 Ordered Twig Pattern Query and Positional Predicates

XML has become a de facto standard to share, save and represent business data over heterogeneous and homogeneous platforms [93, 209, 98]. As a result, large collections of XML data need to be managed and queried efficiently. Popular XML query languages, such as XPath [222] and XQuery [224], provide constructs for specifying relationship patterns serving as structural constraints in XML queries. As been discussed in the above sections, a twig pattern match is the most frequently used query model in the literature

for searching XML [88]. A twig pattern matching problem has been attracted most of the research work in the context of XML querying and indexing [81] since it is seen as equivalent to a subset of XPath which is, in turn, a subset of XQuery. It provides operators to express structures of XML documents in *path expression* of XPath and *path expressions* in **for** and **let** clauses of XQuery [189, 188] (see Chapter 2). XPath can specify more than Ancestor-Descendant and Parent-Child structural patterns, in its specification [222], XPath has thirteen axes and among them only *child* and *descendant*, which represent Parent-Child and Ancestor-Descendant relationships, respectively, appear in a majority of XPath queries in practice. The lists of XPath navigational axes were discussed in Chapter 2, however, very little is known about sibling edges in twig patterns which determine the order of relationships among sibling query nodes. The existing literature on XML twig pattern match is extensive and focuses particularly on processing queries where structural axes are restricted to P-C and A-D. Nevertheless, there is a relatively small body of literature that is concerned with sibling axes in twig pattern queries under branching twig nodes. Table 4.2 illustrates semantics of sibling axes supported by XPath. When these four axes are involved in twig pattern queries, they are referred to as ordered twig pattern queries.

In order to define the query matching for TPQ with order-constraints, the left-to-right relationship plays a vital role in the matching of ordered twig queries. In an ordered twig match, all child query nodes of a branching twig node have to satisfy the left-to-right relationship. Formally, it is as the twig match pre-defined in definition 4.10 with an additional constraint to handle left-to-right relationships or so-called left-to-right ordering among sibling twig query nodes which is the target of the research work of this thesis. It has commonly been assumed that the existing holistic twig algorithms can guarantee nodes contributing in the final solutions by processing a bounded number of elements when twig queries involved A-D axes only, while the existing of P-C edges might lead to the need for processing an unnecessary number of elements, which could be in the order of the size of the input document [218]. In general, this requires the same when processing *following-sibling* and *preceding-sibling* axes. To illustrate the semantics of order-based axes and their corresponding matching process, Figure 4.6 depicts the difference between unordered twig and ordered twig queries over the data tree in Figure 4.6a, of the top of each twig query is the corresponding XPath expression. It should be noted that the order restriction is denoted by a " $<$ " in a box to indicate that all child query nodes of that marked branching query node are ordered from *left* to *right* as in Definition 4.21 [118, 145, 93, 146]. This is a limitation of previous work, in contrast, an ordered branching query node, which is marked by " $<$ ", can have some children which have to be ordered [173].

Definition 4.19 (Left-to-Right ordering or LR ordering)**.** *Considered two nodes* ***u*** *and* ***v*** *in a TPQ which are descendants of a node* ***n****. Let M(u) and M(v) are matching of* ***u*** *and* ***v*** *in D as in definition 4.10. Left-to-Right ordering from* ***u*** *to* ***v*** *is an order constraint specifying that M(u) must appear before M(v) in D, but must not be an ancestor of M(u).*

Property 4.20 (Left-to-Right relationship)**.** *For two nodes u and v encoded in the regional encoding scheme as triples* $(start, end, level)$*, where* $u = (start_u, end_u, level_u)$ *and* $v = (start_v, end_v, level_v)$*. u is to the left of v if and only if*

$$end_u < start_v$$

The following definition is formalised to extend the match of the basic twig query against an XML database, where the structural relationships are parent-child and ancestor-descendant as was defined in the previous section. The above definition (see Definition 4.19) is important to establish the order constraint in ordered twig queries. However, a limitation of this definition is that a query processing strategy based on it can not process more complex ordered twig queries in which ordering between two sibling query nodes may not be significant [146, 118, 145, 146, 188, 230]. For illustration, the XPath expression $a//b[/following :: c]/following :: d$ looks for b-node which has LR ordering with c-node and d-node, it can be seen that the LR ordering between c-node and d-node is not considered since b-node is the context-node associated with order axes (see Figure 4.5).

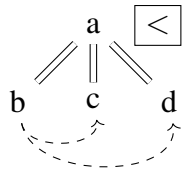


Figure 4.5: Illustration of an ordered twig query with LR ordering. The dashed arrow lines indicate LR ordering between query nodes.

Definition 4.21 (Query Matching)**.** *A match of a twig pattern query* $TPQ = (TP = (V_1, E_1, r_1, \sum_{V_1}, \mu_1), F)$ *in* $D = (V_2, E_2, r_2, \sum_{V_2}, \mu_2)$ *is a total mapping* $M : TP \rightarrow D$ *that maps nodes of TP into nodes of D such that:*

- *For each query node* $u_1 \in V_1$ *maps to* $u_2 \in V_2$ *, then* $\mu_1(u_1) = \mu_2(u_2)$*.*
- *For each edge* $e = (u, v) \in TP$ *where e represents parent-child relationship, M(v) is a child of M(u) in D.*
- *For each edge* $e = (u, v) \in TP$ *where e represents ancestor-descendant relationship, M(v) is a descendant of M(u) in D.*
- *For all query nodes* $u_1, \ldots, u_k \in V_1$ *which are ordered (1 to k) children of an ordered branching twig node* $\mathbf{n}$*, they map to* $v_1, \ldots, v_k \in V_2$*, then* $M(v_1)$ *is to the left of* $M(v_2), \ldots, M(v_{k-1})$ *is to the left of* $M(v_k)$ *in D.*
- *formula F of TPQ is satisfied.*

Example 4.22. *Consider the ordered twig query depicted in Figure 4.6c which represents the path expression* $a/x/following - sibling :: y$*, this twig has one match in the data tree*

Table 4.2: Semantics of sibling axes between two query nodes u and v.

Axis	Semantic
following	u appears in the document before v and v is not a descendant of u.
following-sibling	like following, u and v have the same parent.
preceding	v appears in the document before u and v is not an ancestor of u.
preceding-sibling	like preceding, u and v have the same parent.

of Figure 4.6a as (a_1, x_1, y_2) *as indicated by dashed lines in Figure 4.7. It can be seen from the positional information encoded in document elements, the order between* x_1 *and* y_2 *satisfies the left-to-right ordering as* $2 < 7$*, while* x_2 *and* y_1 *does not satisfy the order constraint as* $5 \not< 4$*. In contrast, the unordered twig of Figure 4.6b in the same data tree has two matches as* (a_1, x_1, y_2) *and* (a_2, x_2, y_1).

XML data are order-sensitive and the need to support querying order model is crucial in many domains. There are many XML models which have to be queried with respect to the order of elements under particular branching elements to preserve the logical structure of the document as in *TreeBank* where the syntactic structure of text data has to be retained when handling queries over it [55]. Early examples of research into OTPQ can be found in a sequence matching approach, where both XML documents and queries are converted to sequences, and query matching, thus, is reduced to subsequence matching. This approach was discussed in the literature review chapter (see chapter 3). Twig pattern matching algorithms based on sequence indexing can support only ordered twig pattern queries due to the nature of sequence encoding of the hierarchical structures of XML documents. Algorithms in this approach are not comparable to holistic twig join algorithms, which is adopted in this thesis, as has been reported [164, 55]. Holistic twig join algorithms are the most robust and predictable solutions when compared with other matching approaches, also they have been regarded as the most efficient group in the literature [164, 209, 24].

In recent years, there has been an increasing amount of literature on *unordered twig pattern* matching where order-based constraints are not significant, whereas there are relatively few research studies in the area of *ordered twig pattern* [55, 145, 118, 90, 218, 146]. The existing twig-based algorithms fail to handle order-based queries efficiently because they concentrate more on P-C and A-D relationships. Thus, they do not impose any order constraints in the processed nodes [173]. In addition to order-based axes, positional predicate, which appears in a twig pattern query as numerical value enclosed within a predicate [2] in the path expression, has been considered in limited approaches as in [217, 70].

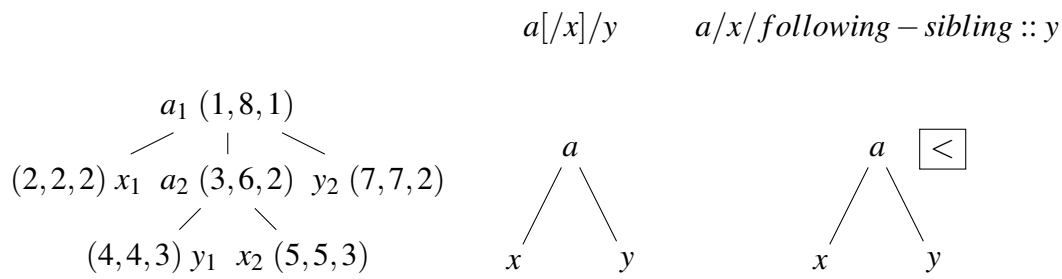
A straightforward technique to answer twig pattern queries containing order-based axes and positional predicates were studied and it might lead to redundant computations

[2]In XPath specification [222], positional predicates can be invoked through pre-defined function called position() inside predicates in the form position() op n, where op is one of the basic logical comparison operators $\{=, <, >, \leq, \geq, \neq\}$and n is an integer. For example path expression to find the second x-node which must be a descendant of a-node which must have y-node as one of its children can be written as $a[/y]//x[position() = 2]$ or abbreviated as $a[/y]//x[2]$

and a large number of intermediate results (i.e path solutions in *TwigStack*) [217]. In order to process ordered twig queries and positional predicates efficiently, the mapped elements have to be checked against theses constraints prior to produce the final solutions. No algorithm can process *order twig pattern query* or positional predicates in linear time. A possible explanation for this may be the lack of a proper mechanism to alleviate *following-sibling* and *preceding-sibling* axes which hold to some extent the same obstacles when processing P-C axis. The representation of XPath expressions with sibling axes as a twig, in this thesis, is based on the ideas presented in [207, 173]. Basically, *preceding* and *preceding-sibling*, or so-called *backward*, axes are converted to their equivalent axes *following* and *following-sibling*, respectively. It has been proven [207] that XPath order-based axes have their natural inverses as: $following^{-1} \rightarrow preceding$ and $following-sibling^{-1} \rightarrow preceding-sibling$, vice versa. One more, processing twig pattern queries efficiently with positional predicates remains a challenge as the occurrences of nodes in XML documents have to be taken into consideration to identify participant elements in final answers to twig queries.

Up to now, there has been no advanced preorder filtering technique, based on *getNext()* function originally introduced in [40], that can filter out irrelevant elements with positional predicates efficiently [70]. Avoiding useless elements can not be handled by top-down filtering strategies as they may result in false negatives (also known as false dismissals or incomplete result). It seems possible that this is due to positional predicates are not permutable. Locations of positional predicates in TPQs might change the semantics of twig queries the matching results. To better understand the processing of twig pattern queries with positional predicates and the semantics of different occurrences of positional predicates, example 4.23 illustrates this point clearly.

Example 4.23. *Consider the XML data tree t_1 of Figure 4.8, the three twigs with positional predicates marked inside brackets next to the branching a-node yield different matching results. The first twig query q_1 asks for the second a-node which is found it has x-node as one of its child nodes if it has child y-node. The match of twig query q_1 on XML tree t_1 returns no match because the second a-node a_2 has only child x-node. While the twig query q_2 questions if the second a-node has children nodes of x-node and y-node, it has no match on XML tree t_1. The last twig query q_3 wants to return the second a-node which already satisfies the structural relationships associated with it, in this example it returns the second a-node that has x-node and y-node. The match of q_3 against the XML tree t_1 yields one match (a_3, x_3, y_3). This can be illustrated briefly by q_2 takes into account the occurrences of elements with the XML tree, q_1 considers the satisfaction of the partial structural relationships (a/x instead of a[x]/y) and the position of the matching result sets is significant in q_3. The different representations model as twigs are achieved by labelling the edges, which have to be satisfied after checking the positional predicates, with "*". They are called post-structural constraints, while the unlabelled edges called pre-structural constraints [70].*



(a) an XML tree t_1. (b) unordered twig query q_1. (c) ordered twig query q_2.

Figure 4.6: The difference between an unordered twig and an ordered twig.

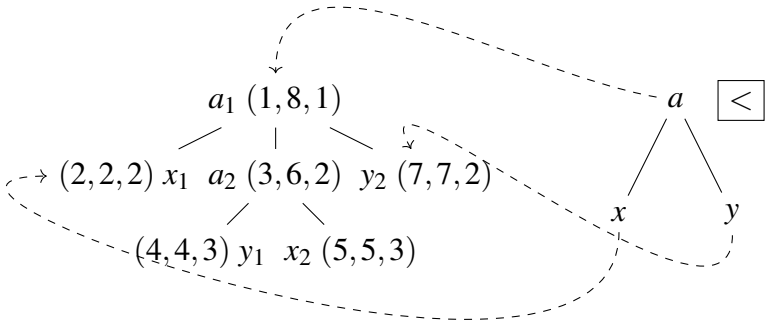
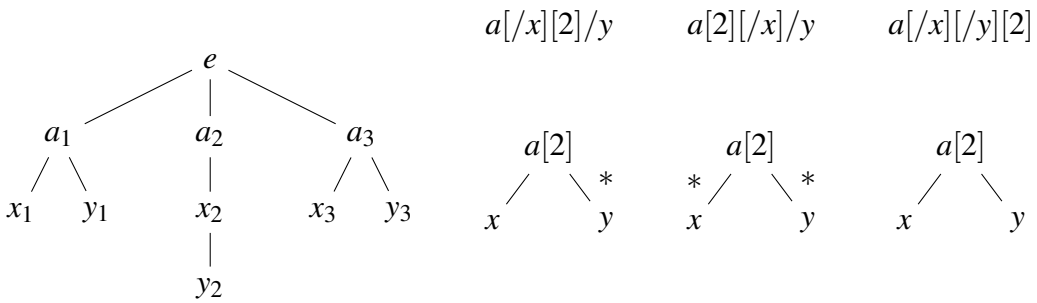


Figure 4.7: Illustration of ordered twig match shown in dashed lines.



(a) an XML tree t_1. (b) a twig query q_1. (c) a twig query q_2. (d) a twig query q_3.

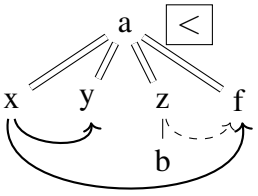
Figure 4.8: A sample of an XML data tree and twig pattern queries with positional predicates. The edges associated with the positional predicates are unlabelled while edges labelled with "*" should be checked after satisfying the positional predicate.

It is important to bear in mind that the previous holistic ordered twig join algorithms have not considered the semantics of order axes in their implementations. In other words, the basic structural relationships among children nodes of ordered twig queries would be processed the same way as *unordered* twig queries while the left-to-right ordering is used to check the order relationship among them to eliminate useless elements. It, therefore, seems that the existing ordered twig algorithms' understanding of the *ordered* twig pattern queries is questionable [55, 145, 118, 90, 218]. In their excellent analysis of order-based axes in XPath expressions, [173] concluded that the semantics of order axes have to be taken into account while encoding path expressions with order axes in twig patterns. This encoding technique would improve the filtering process and identify the participant elements effectively. The existence of *following-sibling* and *preceding-sibling* axes would change the semantics of *ordered* twig queries, according to their sibling query nodes, *ordered* twig queries have to be modified in order to produce the accurate results. The authors extended the definition of left-to-right ordering to handle sibling order axes as in the following definition 4.24. Figure 4.9 provides an overview of LR and SLR ordering in an ordered TPQ.

Definition 4.24 (Sibling Left-to-Right ordering or SLR ordering)**.** *For two nodes* ***u*** *and* ***v*** *in a TPQ are connected to their common parent query node* ***n****. Let M(u) and M(v) are matching of* ***u*** *and* ***v*** *in D as in Definition 4.10. Sibling Left-to-Right ordering from* ***u*** *to* ***v*** *is an order constraint specify that M(u) is and M(v) are siblings in D, and M(u) appears before M(v).*

Property 4.25 (Sibling Left-to-Right relationship)**.** *For two nodes* ***u*** *and* ***v*** *encoded in the regional encoding scheme as 4-tuple* $(start, end, level, parentID)$*, where* $u = (start_u, end_u, level_u, parentID_u)$ *and* $v = (start_v, end_v, level_v, parentID_v)$*.* ***u*** *has a sibling left to right relationship with* ***v*** *if and only if*

$$parentID_u == parentID_v \text{ and } end_u < start_v$$



XPath Expression:

a//x[/following-sibling::y]/following-sibling::f/preceding::z/b

Figure 4.9: Illustration of an ordered twig query with LR and SLR ordering. The dashed arrow line indicates LR ordering, while the solid arrow lines indicate SLR ordering.

The grammar of ordered twig pattern query addressed in this thesis is presented in Figure 4.10. The order relationship can be explicitly declared using one of the four order-based axes which in this case twig queries representing path expressions with these axes might be modified based on the idea in [173]. Otherwise, the order constraints among

$TPQ \rightarrow Pattern$
$Pattern \rightarrow Step(Predicate)^* \ (Pattern)?$
$Step \rightarrow Axis\ NodeTest$
$Predicate \rightarrow [\ Pattern \mid OrderSpec \mid Digits\ ]$
$Axis \rightarrow / \mid // \mid /following :: \mid /following-sibling :: \mid /preceding :: \mid /preceding-sibling ::$
$OrderSpec \rightarrow NodeTest\ NodeComp\ NodeTest$
$NodeComp \rightarrow << \mid >>$
$Digits \rightarrow [0-9]^+$
$NodeTest \rightarrow String$

Figure 4.10: A grammar of TPQ with order-constraints and positional predicate.

children query nodes of branching twig nodes might be imposed inside predicates using node comparison (also known as sequence) operators defined in XPath [222]. These node comparisons take two nodes as operands. To illustrate, $<<$ returns true if the left side operand *precedes* the right hand operand in document order, otherwise it returns false. By the same token, of course, $>>$ mean *follows*. The semantic difference between the two approaches is illustrated in Figure 4.11. This distinction is further exemplified in Chapter 7 in which a new approach to provide an efficient evaluation for TPQs with order axes and sequence operators is proposed. As a result, in an ordered twig match, the query nodes must satisfy the left-to-right ordering, sibling left-to-right ordering and sequence operators. Formally, it is a twig match as defined in Definition 4.10 with the order constraints. Definition 4.28 formalise an ordered twig match proposed in this thesis.

Definition 4.26 (Sequence Left-to-Right ordering or SeqLR ordering)**.** *For two nodes* ***u*** *and* ***v*** *in a TPQ are connected to their common parent query node* ***n****. Let M(u) and M(v) are matching of* ***u*** *and* ***v*** *in D as in Definition 4.10. Sequence Left-to-Right ordering from* ***u*** *to* ***v*** *is an order constraint specify that M(u) appears before M(v) in document order.*

Property 4.27 (Sequence Left-to-Right relationship)**.** *For two nodes* ***u*** *and* ***v*** *encoded in the regional encoding scheme as 4-tuple* $(start, end, level, parentID)$*, where* $u = (start_u, end_u, level_u, parentID_u)$ *and* $v = (start_v, end_v, level_v, parentID_v)$*.* ***u*** *has a sequence left to right relationship with* ***v*** *if and only if*

$$start_u < start_v$$

It is possible, therefore, that a novel combination of previous approaches to handle parent-child axes can be exploited to process order axes, this could be achieved by proposing a new mechanism which gives efficient evaluation for twig pattern queries with Parent-Child edges and the existing techniques can then be used to evaluate order axes. An efficient combination might lead to algorithms processing twig pattern queries with order axes efficiently. Moving on now to consider the improvement on *top-down* and *bottom-up* combinations to improve twig join algorithms. In the following section, the limitations and the advantages of combining top-down processing with bottom-up filtering will be discussed in order to demonstrate further possibilities for filtering twig pattern queries involving P-C edges.

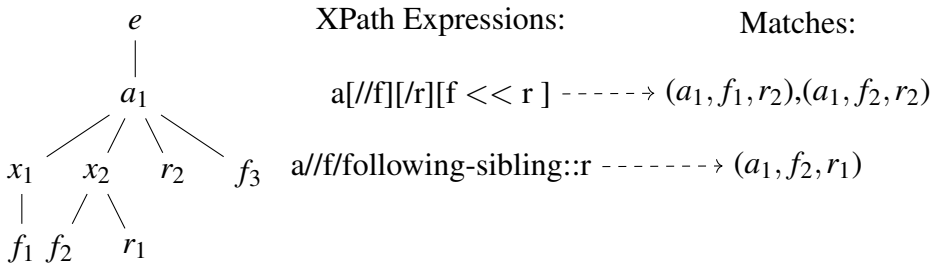


Figure 4.11: The semantics of order constraints imposing in two different ways. In the first path expression using << operator, the processing can be made using the existing algorithms with post-processing operation to prune false positives regarding document order, while the second expression requires a modification to the structural relationship between a-node and r-node in order to produce the accurate result. The dashed arrow lines indicate the query matches.

Definition 4.28 (Query Matching)**.** *A match of a twig pattern query* $TPQ = (TP = (V_1, E_1, r_1, \sum_{V_1}, \mu_1), F)$ *in* $D = (V_2, E_2, r_2, \sum_{V_2}, \mu_2)$ *is a total mapping* $M : TP \rightarrow D$ *that maps nodes of TP into nodes of D such that:*

- *For each query node* $u_1 \in V_1$ *maps to* $u_2 \in V_2$ *, then* $\mu_1(u_1) = \mu_2(u_2)$.
- *For each edge* $e = (u, v) \in TP$ *where e represents parent-child relationship, M(v) is a child of M(u) in D.*
- *For each edge* $e = (u, v) \in TP$ *where e represents ancestor-descendant relationship, M(v) is a descendant of M(u) in D.*
- *For each query node* $u_1 \in V_1$ *which is a child of an ordered branching twig node* ***n****, if* u_1 *has LR ordering with sibling query nodes* $u_n, \ldots, u_m \in V_1$*, then* $M(u_1)$ *is to the left of* $M(v_n), \ldots, M(u_1)$ *is to the left of* $M(v_m)$ *in D.*
- *For each query node* $u_1 \in V_1$ *which is a child of an ordered branching twig node* ***n****, if* u_1 *has SLR ordering with sibling query nodes* $u_n, \ldots, u_m \in V_1$*, then* $M(u_1)$ *is the sibling left of* $M(v_n), \ldots, M(u_1)$ *is the sibling left of* $M(v_m)$ *in D.*
- *For each query node* $u_1 \in V_1$ *which is a child of an ordered branching twig node* ***n****, if* u_1 *has SeqLR ordering with sibling query nodes* $u_n, \ldots, u_m \in V_1$*, then* $M(u_1)$ *is the sequence left of* $M(v_n), \ldots, M(u_1)$ *is the sequence left of* $M(v_m)$ *in D.*
- *formula F of TPQ is satisfied.*

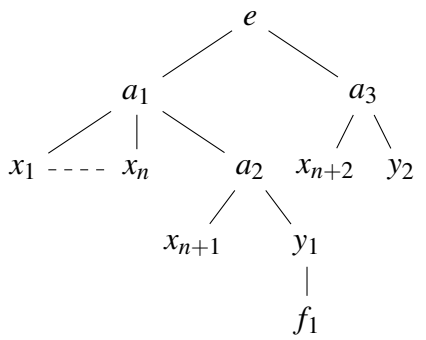
4.2.3 Combination of Different Filtering Strategies

An XML technology has emerged as de facto standard for storage of semi-structure data and for data exchange in e-business [88]. TPQ is a core operation in XPath [222] and XQuery [224] which are popular of XML query languages. TPQ represents *path expression* which is the basic building block of XML query languages. The existing literature on XML

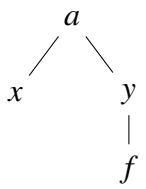
query processing is extensive and focuses particularly on twig pattern matching problem [209]. Several twig join algorithms have been proposed to search XML data. As was pointed out in the literature review, the existing twig join algorithms have been grouped into two main lines of improvements over the pioneering holistic twig join algorithm, *TwigStack* [40]. Top-down twig join algorithms process TPQs by reading the nodes in pre-order traversal of the input document and checking children solution extensions for internal query nodes. The second category is bottom-up algorithms which store elements of the input document in post-order manner and inspect matching elements through virtual sub-trees, one major drawback of this approach is that high memory consumption due to the fact that all elements mapping to leaf query nodes reside in the main memory until the entire trees are completely processed. Although, they have better practical performance than algorithms using top-down processing.

In the literature, top-down processing, which is based on *getNext()* [40], has been associated with bottom-up algorithms as a filter in order to reduce memory usage and improve thus the overall performance. As discussed in Section 4.2.1, the main weakness of the top-down filter is the failure to provide an optimal evaluation for TPQs with parent-child relationships. Advantages of top-down filter in bottom-up algorithms can be broken down into speeding up the sequential reading of the input streams and avoiding the storage of elements which do not have ancestors likely participating in the final solutions. No top-down filter algorithm can remove leaf query nodes effectively when a mixed of P-C and A-D queries are processed. Recently, two twig join algorithms, which are proposed in [22, 89], have been considered as the most superior ones which improve top-down and bottom-up combinations. The authors of [89] proposed a new advanced preorder filtering function called *getPart()* which introduces two improvements of the original *getNext()* function. Compared with the existing preorder filtering function, *getPart()* returns only child query nodes if and only if they satisfy weak prefix filtering checks with their parents. The second improvement is to perform a cursor forward movement according to the current query node descendants and ancestors, in contrast to the *getNext* which performs a cursor forward movement according to the current query node descendants. Other authors [22] further improved the *getPart()* function by avoiding unnecessary function calls. The new advanced preorder filtering function is called *getMatch*. The *getPart()* function is served as the advanced preorder filtering strategy for a family of twig matching algorithms devised in [89] whereas the GTPStack algorithm which is proposed in [22] uses the *getMatch()* function. Figures 4.13 and 4.14 illustrates the difference between these two approaches. It can be seen in this examples lower query nodes, specifically nodes $x_1, \ldots, x_n$, are useless, they though processed and stored in the intermediate storage. There is some evidence to suggest that a strong filter which can prune irrelevant elements corresponding to internal and leaf query nodes would improve the practical performance [88]. Filtering strategy is optimal for a TPQ if it skips all irrelevant nodes during sequential read of the input

document, and the twig query processing algorithms use this filter has a linear worst-case I/O complexity with respect to the sum of the input and output sizes for TPQ [198, 22].

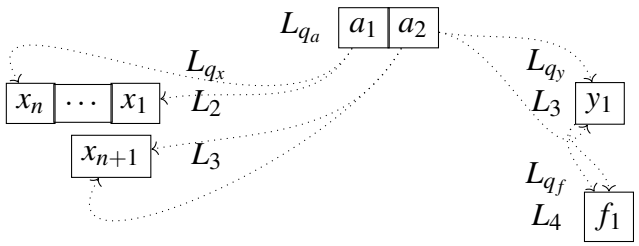


(a) an XML tree t_1.

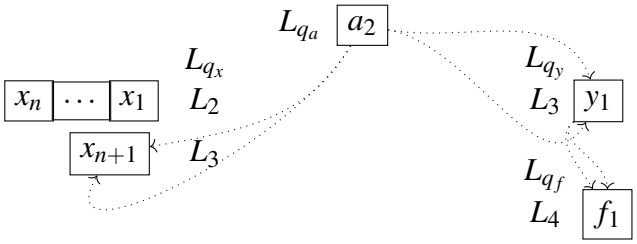


(b) a twig query q_1.

Figure 4.12: Inefficiency of top-down filter strategy.



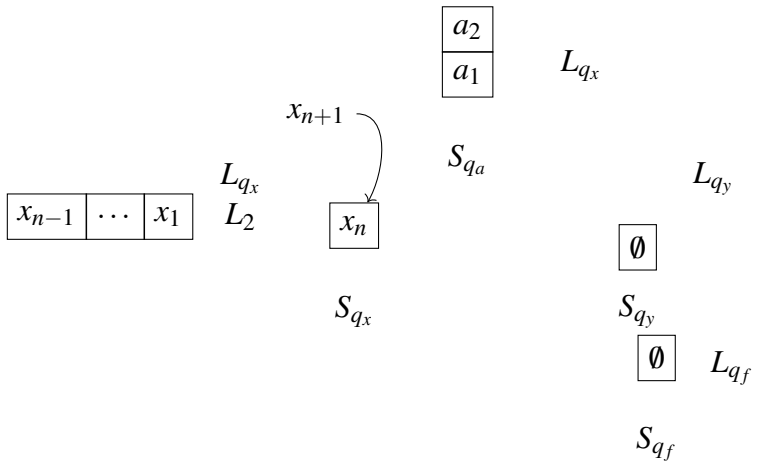
(a) The intermediate storage state when all elements have been processed.



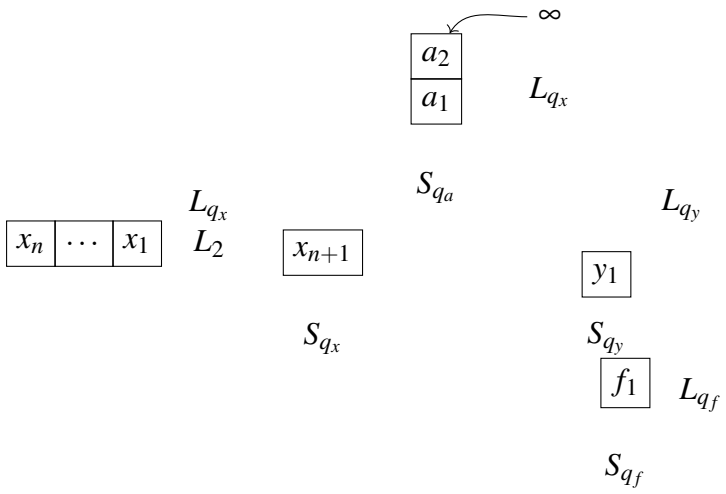
(b) The intermediate storage after performing strict bottom-up filter using level split vectors.

Figure 4.13: TJStrictPre algorithm state when evaluating the twig query q_1 against the XML tree t_1 of Figure 4.12. The interval pointers are shown in dashed lines.

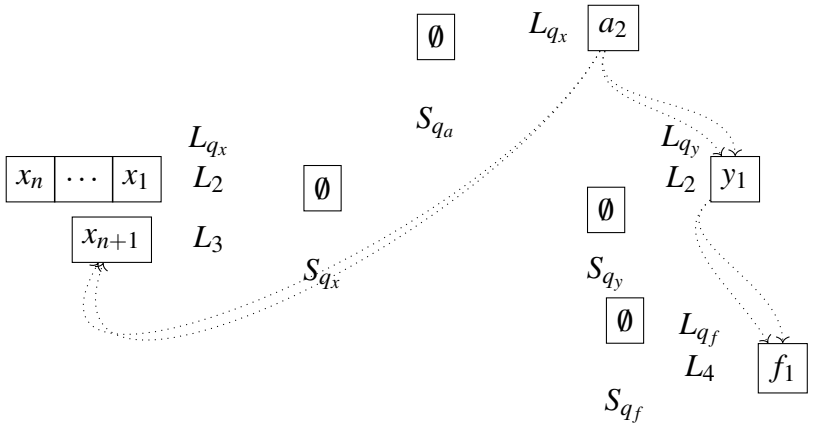
An advanced preorder filter, which is used in top-down approaches, reads the nodes in document order, then passes them to the main algorithm which pushes them into stacks for further processing and causes them to pop out in an order which does not correspond to the document order nor the document postorder. Note that only nodes corresponding to the same query node are popped out in the document postorder because the main data structure used in XML query processing algorithms is a *stack*, and there is some evidence to suggest that every holistic twig join algorithm uses stacks during query processing [22]. While postorder algorithms require nodes to be popped out in the document postorder.



(a)



(b)



(c)

Figure 4.14: Illustration to GTPStack operations for the data tree and the twig pattern query of Figure 4.12. (a) GTPStack right before x_{n+1} is about to be processed. (b) GTPStack after all elements has to be read. (c) the final intermediate storage read for enumeration.

Taken together, this is an obstacle when a combination of preorder and postorder filters are proposed. A node processing order (also known as a node pushing order) is essential to

Table 4.3: Classification of holistic twig join algorithms according to their node processing order.

Node processing order	Holistic twig join algorithms
Global process order	PathStack [40], Twig^{2}Stack [53], TwigList [185], TJStrictPost [89] and TwigPos [70]
Local process order	TwigStack [40], TwigFast [132], TJStrictPre [89] and GTPStack [22]
Unclassified	TwigStackList [144]

return answers correctly to twig pattern queries. There are two types of node processing order in the literature:

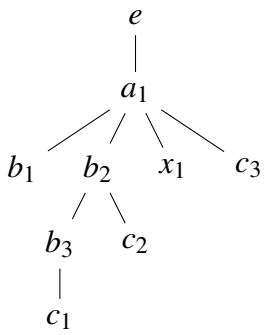
1. *Global process order* in which a node n_i is processed (or pushed onto a stack) before a node n_j if n_i *precedes* n_j in the document order.

2. *Local process order* in which a node n_i is processed (or pushed onto a stack) before a node n_j if n_i *precedes* n_j in the document order and n_i corresponds to the parent query node of n_j.

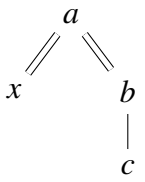
There is an approach where nodes are processed in an order which corresponds neither to global push order nor local push order as in *TwigStackList* [144]. It seems possible that this is due to the buffering technique used which violates the document order even if nodes corresponds to the same query node. This means *TwigStackList's* filter strategy can not be applied to postorder algorithms because nodes can not be popped out in the order they require [132, 22]. On the other hand, a node pop order is determined by the node push order and the pop sequence operation used in holistic algorithms [89, 22].

A *global pop order* can be defined analogously to the definition of *global push order*. However, a *local pop order* pops nodes in the document postorder only when nodes correspond to the same query node. It has been shown that a local push order can not be followed by a global pop order as was assumed in a previous approach which lead to some results are lost [89]. Consequently, the work of [89] has modified the *local push order* to ensure nodes are pushed in a strict order corresponding to the document order which can then work accurately with postorder algorithms. In [22], the researchers proposed a new pop sequence operation (called relaxed pop order) in which a node is popped out after popping out its all descendants when a *local push order* is used. Neither approach guarantees a *global pop order* but they are sufficient to determine *subtree matching* (see Definition 4.14) in postorder manner. Table 4.3 provides the breakdown of holistic twig join algorithms according to their node pushing orders.

Example 4.29. *Consider the XML tree in Figure 4.15 and the query* $//a[//x]//b/c$*, Figure 4.15c shows different push order sequences during holistic processing and the corresponding node pop order sequences are presented in Figure 4.15d. It can be seen from the tables that a combined approach can be problematic and it must take into consideration the node pushing and popping order in order to process the query correctly, otherwise some results might be missed.*



(a) an XML tree t_1.



(b) a twig query q_1.

Global push order	$a_1\ b_1\ b_2\ b_3\ c_1\ c_2\ x_1\ c_3$
Local push order	$a_1\ b_2\ b_3\ c_1\ c_2\ c_3\ x_1$
TwigStackList's push order	$a_1\ b_3\ c_1\ b_2\ c_2\ c_3\ x_1$
Strict push order	$a_1\ b_2\ b_3\ c_1\ c_2\ x_1$

(c) a node push order.

Global pop order	$b_1\ c_1\ b_3\ c_2\ b_2\ x_1\ c_3\ a_1$
Local pop order	$c_1\ b_3\ b_2\ c_2\ c_3\ x_1\ a_1$
TwigStackList's pop order	$b_3\ c_1\ b_2\ c_2\ c_3\ x_1\ a_1$
Relaxed pop order	$c_1\ b_3\ c_2\ b_2\ x_1\ a_1$

(d) a node pop order.

Figure 4.15: Different node push order and node pop order sequences in the literature..

In the literature, holistic query processing for TPQs may be divided into three phases. The first phase in which elements are read from indexed XML documents and node filtering is then performed. Lastly potential nodes are stored in the intermediate storage and an output enumeration is computed. Several approaches have been proposed to optimise the node reading and filtering phases [88, 147, 144, 53, 125, 181, 89]. The existing optimizations include the utilization of indexed streaming lists [40], making use of a refined access method [54, 18, 21], the utilization of path labelling scheme [147, 146, 125, 19], and performing node buffering techniques [144]. These optimization techniques are orthogonal to postorder algorithms with the sole exception of node buffering techniques in which nodes can not be in either global or local pushing (popping) order which gives uncertainty of the node order of the result. The main advantage of node buffering techniques is that they can filter out irrelevant nodes efficiently using a simple *tag streaming scheme*. It can thus be suggested that an improved tag streaming preorder filtering function may reduce the combined approach's processing time.

4.3 Research Methodology

In an academic context, a generally accepted definition of scientific research is a systematic process of collecting, analysing and interpreting data with the objective of discovering and disseminating new knowledge [63]. The research process is systematic in that it makes use

of rigorous methodology for achieving the objective of research [234, 29, 200, 99, 193, 14]. A well-defined explanation of research process would result in a "good research" as [200] said, it is thought that acceptance of research's results relies on the process of acquiring and analysing the results [200, 14, 193]. The work of [99] proposed guidelines which describe characteristics of well-carried out research in Information System discipline. Generally, the research philosophy (or so-called paradigm) explains how the source influences the researcher and how the researcher deals with the development of new knowledge [63, 69]. It identifies the trustworthy evidence to answer research questions or hypotheses.

Within the scope of Computer Science discipline, there are four dominant research philosophies: Positivism, Interpretivism, Critical Theory and Pragmatism [63, 29, 234, 158]. The research work conducted in this study adopts the positivism research philosophy. Positivism is the most suitable research philosophy for this study because a positivism study determines if there is a cause-effect relationship as a means of representing the need to identify and assess causes that influences outcomes [63, 69, 158]. Consequently, the current work examines how the amount of information contained within XML labels can be increased in order to implement a set of holistic twig join algorithms which may have the potential to improve the performance of XML twig query processing. Furthermore, quantitative research is generally associated with the positivist research philosophy. Thus, theory which is the building block of scientific knowledge is seen in quantitative research as the use of interrelated set of variables formed in the hypothesis (see Section 4.3.2) to specify the relationship among them in order to predict outcomes. Science as an underlying ground of positivism is the process of verifying theories by testing hypotheses derived from them [158, 14, 216, 233]. Throughout this thesis, hypotheses are tested and verified based on the deductive inference in quantitative research in which hypotheses are the main route to draw logical conclusions and form specific theories. The current work, thus, develops a set of hypotheses in order to assess the effect of information contained within XML labels on improving the efficiency of holistic twig join algorithms using a deductive approach.

Quantitative approach can begin with at least one research question, which is about the central phenomenon being explored [63, 159, 158]. Quantitative researchers need to adopt methodology to test their hypotheses. Researchers in Computer Science use various methodologies to solve research questions within the discipline, among these methodologies *experimental methodology* is the most common [14, 216]. It is believed that Computer Science is also experimental science [104]. It could be argued that the research questions are necessary to have a clearer understanding of the phenomena as well as the rational basis for selecting the appropriate methodology. Section 4.3.1 will highlight research questions (the nature of problem) that this research work is dealing with. The most important process for the successful completion of research project is the choice of the appropriate research methodology which generally guarantees a systematic approach to carry out a research. The research methodology used to test hypotheses for their validity in this thesis is discussed in Section 4.3.2.

4.3.1 Research Questions

Having identified the research problems (see Section 4.2), researchers may need to form exploratory questions which help them to choose the appropriate research methodologies [69]. Analysing problems and clarifying research questions are thus considered a fundamental first step to help in the selection of the appropriate methodology [210, 158]. In this study, the general research question addressed here is:

> It is possible that inserting more meta data inside XML labels may further improve query response time and reduce main memory consumption cost with minimal computation and space overhead.

Basically, this question identifies two main issues: efficiency and scalability in the context of XML twig pattern matching. To better understand the research question, it has been refined into more specific questions.

1. **What meta-data could usefully be inserted without a significant effect on label storage?**

   Following the discussion in Section 4.2.1 and by being aware of the main research problem in XML query processing models, an optimal XML twig matching algorithm should be able to filter out useless elements which do not contribute in the answers to twig pattern queries. Over the past decades, several approaches have been proposed in the literature to handle P-C relationship efficiently and provide an optimal evaluation for twig patterns with P-C axes. The utilization of refined inverted indexing schemes based on the depth information of XML elements have been considered, and the proposed schemes have shown a superior performance in processing only P-C relationship in all twig patterns' edges. Grouping the XML elements according to their unique labelled paths has been proposed in [54] which is resemble to the structural index generated by strong *DataGuide* [85, 21]. The proposed prefix path labelled streaming scheme has been proven [22, 21, 18] to prune irrelevant elements efficiently. However, there are still twig queries where both *tag-level* and *prefixed path labelled* can not guarantee optimal evaluations [22, 18], and the only indexing technique which is able to cover all twig queries is *F&B-Index* proposed in [119]. The elements of XML documents are grouped in partitions according to the indexing technique used, they are sorted in *document order* and encoded using a labelling scheme. Therefore, the labels of elements within an XML document can be exploited to combine the advantages of different partition techniques to facilitate the structural relationship determination.

   Another important finding was that in [217, 207], the authors exploited the information of parent elements for two or more context-node under an investigation to speed up the process of handling sibling-axes in ordered twig queries. They modified the original range-based labelling scheme to accelerate the following-sibling relationship processing by incorporating the *parentID* which is the *start* filed of

the parent element in the original labelling scheme. As a result, each element is encoded a quadruple as *(start,end,level,parentID)*. In addition, it was hypothesised that registering the information of the total number of children under branching elements has provided a significant mechanism to identify contributing branching elements and prune irrelevant elements while scanning the input streams [125]. The authors stated that the dominant cost of XML query processing is the *data access* cost. In this study, they augmented the extended Dewey labelling scheme proposed in [147] with their novel approach to reduce I/O overhead. It is possible, therefore, that recording the distinct tags of child elements under internal elements would be more useful to filter out useless elements without incurring significant costs.

2. **How can the meta-data of XML labels be extended to alleviate some issues in XML query processing?**

   In practice, XML documents may be very large, complex and have deeply nested elements [147]. Therefore, a straightforward approach to incorporate additional information within XML labels is to use bits representing element names or number of children of internal elements. The key problem with this approach is that the label size is very large. There are several optimization techniques which propose more promising labelling scheme to overcome this limitation [147, 125, 199]. However, they are too expensive to be used for processing deeply recursive data trees as the space usage increases and internal nodes can not be guaranteed to participate in final results. One major drawback of these approaches is that they rely on the existence of schema information in which elements' names are derived and the children tags information is computed. The proposed labelling schemes can be generated by scanning the document in a tree-traversal order at least twice. Moreover, holistic algorithms based on a refined labelling scheme with extra information require the maintenance of additional indices and data structures during query processing such as *finite state transducer* [147] and *pointer structure* [125]. One of the limitations with the work of [199] is that it does not explain how it can derive child name information during query processing. It may be that proposing a technique, which is schema-less as the presence of schema is not required, to extract the distinct child name while scanning XML documents in depth-first traversal order only once would be more likely to improve the efficiency of XML tree pattern query processing. Depth-first traversal scanning of XML trees can be carried out efficiently using the outstanding event-based XML parser *SAX* (the Simple API for XML) [194].

It should be noted that research questions state what the researcher wants to learn. Hypotheses, in contrast, are perceived to be tentative answers to these questions [29, 69]. The above research questions serve as a mix of knowledge question and design questions to provide sufficient information about the nature of problem addressed in this thesis. Having specified the research questions, the next step is to formulate a research hypothesis and identify the research methodology applied in this research.

4.3.2 Research Hypothesis

Motivated by a critical investigation of existing XML query processing approaches (see Chapter 3), and following on the previous discussion in the above sections (see Sections 4.2.1, 4.2.2 and 4.2.3), this research aims to test the following hypothesis:

> **"Encoding names of child elements of branching XML's elements based on Parent-Child relationships may improve the efficiency of holistic twig join algorithms by increasing the query processor's coverage as well as reducing computation cost and memory consumption."**

Hypotheses must be tested rigorously using the appropriate methodology [65, 29, 63]. A methodology may be defined as a system of principles around which empirical data is collected and analysed [69, 179, 193, 200, 216, 158]. Throughout this thesis, the above hypothesis is tested using an experimental methodology. Experimental methodology (also known as experiment or controlled experiment) is widely used in Computer Science to evaluate new solutions [99, 179, 158, 14, 216]. It involves an iteration of hypothesize and test process. Also, experiment is the most appropriate for this study because it is about an investigation of a testable hypothesis in which one or more *independent* variables receive the experimental manipulation to measure their influences on one or more *dependent* variables [63, 69].

A precondition to experiment is a clear hypothesis. The above hypothesis decides what variables to include and identifies clearly *independent* and *dependent* variables [193]. As a result, the present study is concerned with understanding the behaviour of holistic twig join algorithms as a means of improving the performance of XML query processing by providing an augmentation into the existing labelling schemes with minimal computation and space overheads. This means that a novel set of holistic twig join algorithms based on the **Child Prime Label** approach (see Chapters 6, 7, 8 and 9) are the *independent* variables, and the *dependent* variables should be the query processing time and memory consumption. On the question of developing a new theory, evaluating the experiments (see Chapter 10) which are used to test the hypothesis and their results should help in generalising a new theory and identifying contributions and limitations of this study [63, 69, 208]. It should be noted that this study as quantitative research involves data collection that is numerical and subjected to statistical procedures in order to support or refute hypotheses [234, 104]. This research thus provides conclusions with statistical significances.

To conclude this section, the hypotheses in this research will be tested and assessed by a practical implementation that is designed to evaluate them. More precisely, the main aim of this study is the processing and optimization of XML twig pattern matching. The implementation covers many aspects:

- Parsing, labelling and partitioning XML documents.
- Providing stream management systems.

- Implementing twig join algorithms which are previously proposed in the literature.
- Developing twig join algorithms which are originally proposed in this thesis (details in this will be discussed in Chapters 5).

4.4 The Scope of the Research

Many different twig join algorithms for XML TPQ processing have been proposed in the literature on XML twig join processing (see Chapter 3). Performance of holistic twig join algorithms [40, 53, 185, 132, 89] has been proven to be superior to other approaches [188, 230, 5, 105], particularly when the XML tree is deeply recursive [173]. Holistic approaches are considered the most robust solutions which do not require further complicated query optimizations [18]. Hence, TPQ processing complexity can be determined efficiently when holistic approaches are deployed because of their optimal processing of a wide range of XML query types. The main aim of this study is to propose a comprehensive set of holistic twig join algorithms that consume less time and memory than the existing approaches. The performance of the proposed holistic algorithms will be tested based on the analysis of two significant factors, which are the most frequent in the literature of XML query processing, namely: query processing time and number of nodes processed.

The work of this thesis investigates the factors that improve the efficiency of holistic algorithms regarding the worst-case I/O complexity of holistic twig join algorithms. This study also provides new insights into the link between the information contained in XML labels and the efficiency of XML twig join algorithms in terms of processing time capabilities and memory consumptions. In this thesis, analysis of the state-of-the-art holistic approaches is carried out in order to propose additional improvements. Thus far, the thesis focuses mainly on node filtering optimisations, which are the main challenge faced by many holistic approaches, performing a structural match. The purpose of this section is to outline the scope of this thesis. The next section describes the research objectives acquired when applying the hypothesis formalised in Section 4.3.2.

4.5 The Main Objectives for the Solution

Having identified the research problems emerging from the analysis of XML processing in Section 4.2 and deploying the research hypothesis stated in Section 4.3.2 which results in the proposal discussed in Chapter 6, this research work has two significant objectives which are explained in the following sections. As was mentioned in the research methodology section (see Section 4.3), the work in this thesis revolves around the hypothesis formalised in this chapter but it is not exclusively basing on it. In other words, the main basis for proposing comprehensible process model for twig join algorithms is obtained from the accumulated findings throughout the experiments. Accordingly, the hypothesis will be

expanded to cope with the findings which emerge from this study. As a result, contributions of the thesis will be discussed later in Chapter 11.

4.5.1 Extending the Existing Labelling Schemes

XML documents organize data in hierarchical structures and describe semantic relationships among data elements by user-defined tags. One of the most important features of labelling scheme is that it can be used to identify structural relationships efficiently without the need to access the entire documents [147, 253, 90, 238, 243, 244, 140]. The names of XML data elements (also referred to as tag names) have not been exploited effectively to capture structural relationships involving more than two elements which share the same parent. This is an important issue for improving the efficiency of holistic twig join algorithms. The proposed approach may prove to be particularly valuable to process different types of twig pattern queries which contain axes rather than the Ancestor-Descendant and Parent-Child axes introduced in the XPath specification [222].

4.5.2 Improving the Structural Match of TPQ

Structural match is a fundamental property of XML query which is a combination of structural search and content search. Processing both structural and value searches on XMl databases requires the determination of structural relationships between elements and fast access to the desired content. Evidence suggests that structural match is among the most important factors for speeding up the process of XML queries [103]. The majority of XML TPQ algorithms utilize twig representation of XPath path expressions since twig structure is orthogonal to tree-structured adopted for XML data [173, 93, 209, 18, 22]. The proposed approach can be integrated with the existing holistic algorithms in order to facilitate query processing performance. Novel combinations of previous approaches can be devised around the new labelling scheme in which suboptimal processing of TPQs with a combination of A-D and P-C edges in terms of running time can be expected to be superior when compared to the existing work. The new approaches also can be expected to process TPQs containing order axes efficiently by capturing the semantics of order constraints involved.

4.6 Conclusion

Finding all occurrences of TPQs in XML documents is considered as a specific task for XML query processing of XML databases. There is room for improvement as suboptimal processing still exists as was explained in Section 4.2. Improving labelling schemes is vital to optimise node filtering strategies of holistic algorithms and process different axes of XPath [222] efficiently.

This chapter aims at highlighting the research problems and motivations and describing the research methodology adopted in this thesis in order to ensure the systematic process for carrying out scientific research. The research questions were asked to provide a clear understanding of the problem. Then, the research hypothesis, which arises from the literature review (see Chapter 3) and the research problems identified in this chapter, were described. The research hypothesis has been formulated by proposing an augmentation into the existing labelling schemes in order to improve the efficiency of holistic algorithms in terms of processing time and space overheads. The scope of this study and the potential research objectives were discussed as well. The next chapter describes the overview of experimental framework design for XML query processing model used in this study.

Chapter 5

Experimental Framework

5.1 Introduction

This chapter explains the general framework used in this research study to conduct a set of experiments which aims mainly to evaluate the research hypothesis stated in Chapter 4. These experiments will be described in detail as parts of the testing and evaluation processes regarding their functionalities (see Chapters 6, 7, 8 and 9). Each experiment is designed to test different aspects of holistic twig matching algorithms and derive specific goals. As was mentioned in Chapter 4, The main aim of this study is to improve the performance of holistic twig matching algorithms and increase query-coverage of XML query processors adopting a holistic model [40, 61]. Thus, this chapter describes the overall test environment in order to evaluate the properties of the algorithms proposed.

The holistic model [146, 40, 22, 147, 89] for processing XML queries consists of two stages. The first stage can be considered as an off-line stage which performs three basic operations: XML parser, labelling and partitioning and tag indexing (this will be further discussed in Chapter 6). The on-line stage handles basic steps in query processing over indexed XML data. The complete design of the holistic model used in this thesis is illustrated in Figure 5.1. Note that *Tag Indexing (TI)* is an idea originally proposed in this thesis and is incorporated into the holistic model to facilitate the determination of Parent-Child edges. It will be discussed in full detail in Chapter 6. There is no existing standard platform in which holistic twig matching algorithms can be assessed in the same way to compare fairly their performance. Therefore, the main purpose of this chapter is to describe the complete architecture of the experimental framework to overcome this issue. Consequently, each holistic twig matching algorithm utilised in this framework is implemented from scratch in order to calibrate machine speeds and have all the benchmarked algorithms compared based on their characteristics [202, 156]. In addition to the discussion in this chapter, the discussion from the later chapters, in which new approaches to process XML twig queries are proposed, can be combined to facilitate the implementation and design of experiments to be applied to the new approaches.

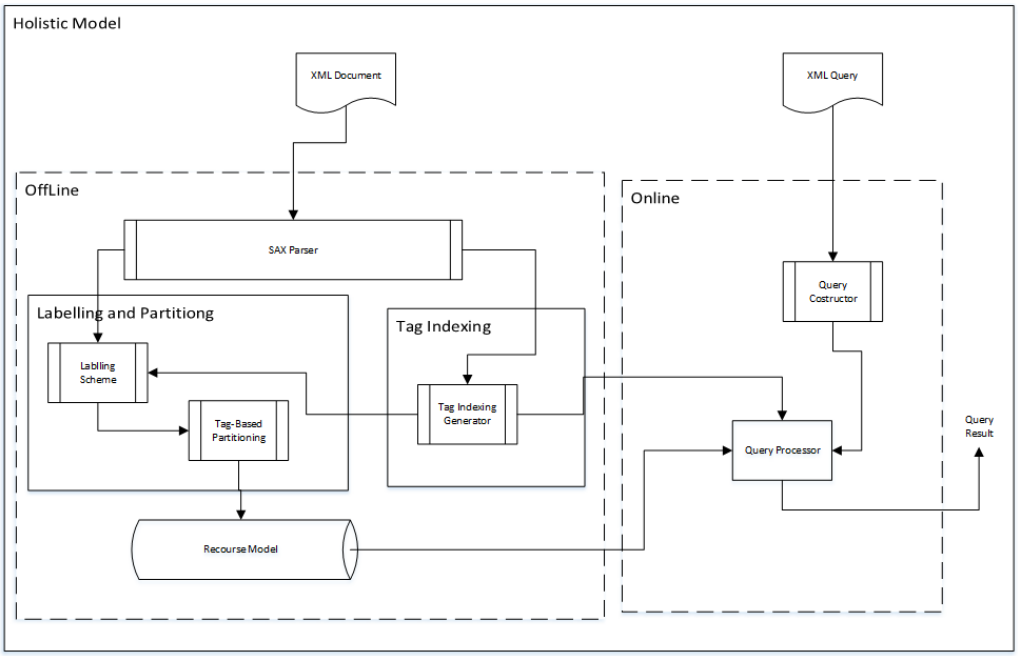
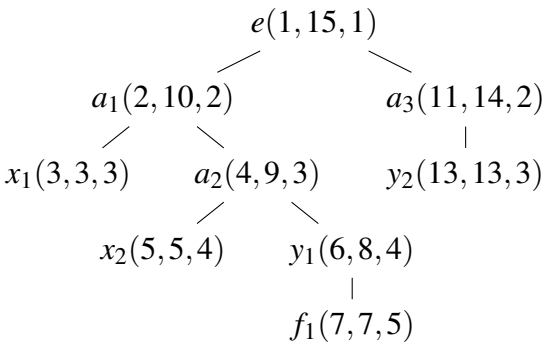


Figure 5.1: The framework of holistic model.

The rest of this chapter is structured as follows. Section 5.2 presents an informal description of the design of holistic model for query processing in native XML databases, its basic components and its features. The implementation of the experimental framework in Section 5.3 An overview of the existing benchmarked XML datasets used in the literature will be discussed in Section 5.4, the selected XML datasets in the experiments are described as well. Next, the statistical analysis procedures used to analyse the collected data are discussed in Section 5.5. Finally, Section 5.6 concludes this chapter.

5.2 Holistic Model Overview

This section illustrates the main components of XML query processing adopting *holistic Model* (HM) [40]. The complete design of HM is depicted in Figure 5.1. It is necessary here to clarify exactly what is meant by *holistic model* from XML query processing perspective. There are two main approaches for twig pattern query processing in XML databases, namely navigational and join-based. The first approach is considered to be the native one inherited from relational databases [34]. In this approach a query plan relies on scan operations on a document index rather than a partition index (see Chapter 3). Figure 5.2 illustrates the difference between these two approaches. The underlying data storage is analogous to DOM-like tree [225] in main memory in order to allow the traversal of XML documents sequentially through pointers to find match of TPQs [20]. The latter approach, join-based, relies on a partition index in which nodes are stored in disjoint groups. The join-based approach can be further classified into two main categories. The first category decomposes twig pattern queries into a set of binary structures, and searches occurrences of TPQs by performing consecutive binary structural joins (see Chapter 3) [5]. The holistic



(a) an XML tree t_1.

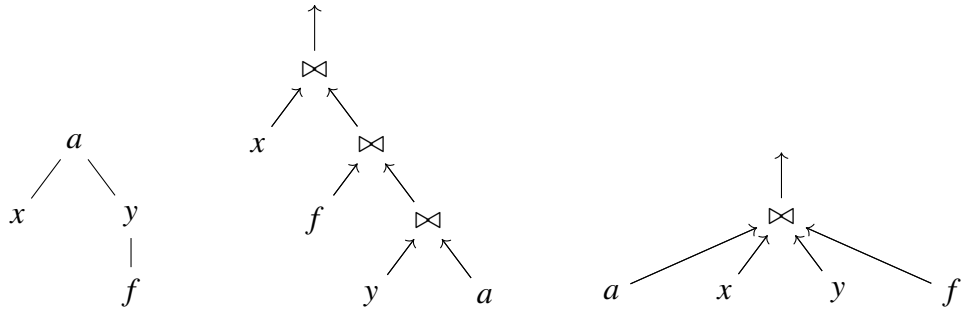
Tag	Label
e	(1,15,1)
a	(2,10,2)
x	(3,3,3)
a	(4,9,3)
...	...

(b) document index.

Tag	Label
e	(1,15,1)
a	(2,10,2) (4,9,3) (11,14,2)
x	(3,3,3) (5,5,4)
y	(6,8,4) (13,13,3)
f	(7,7,5)

(c) partition index.

Figure 5.2: A fragment of document and partition index for a sample of XML tree.



(a) a twig query. (b) a binary structural join. (c) a holistic structural join.

Figure 5.3: A holistic join versus binary structural joins from query plan point of view.

approach , in contrast, uses input streams corresponding to query nodes and processes the entire TPQ by using one holistic operator as opposite to extensive structural joins carried out in the binary structural approach [40]. Figure 5.3 demonstrates the difference between these two approaches from query plan perspective.

To study the performance of holistic twig matching algorithms over indexed documents, it is very important to demonstrate the architecture which underlines query evaluation. Therefore, the *holistic Model* can be divided into storage and execution models as illustrated in Figure 5.1. The storage model includes the process of parsing XML documents to derive the logical structures and indices which deploys node labelling schemes, partitioning mechanisms and *tag index generator*. The query execution model coordinates two main operations. In the first operation, query constructor's result, which is the twig representation of the input XML query, is passed to the next operation to handle twig pattern queries. Query processor contains a set of holistic twig matching algorithms to process twig pattern queries over indexed documents. The following sections describe the primary components of the *holistic Model*. The storage model is discussed in Section 5.2.1; whilst the execution model is illustrated in Section 5.2.2.

5.2.1 Storage Model

In XML DBMSs (XDBMS, for short), fetching data is one of the dominant cost drivers for XML query evaluation, hence this section describes how XML data is stored and how it can be accessed in order to facilitate the efficient processing of TPQs in the query evaluation phase (the system architecture is depicted in Figure 5.1) [7, 34]. This stage can be broadly defined as an off-line pre-processing stage which is designed to prepare information needed in the next stage to process TPQs in holistic way. It consists of four processes each of which conducts a set of operations for parsing,labelling, partitioning and indexing the underlying XML data. The roles of these processes are summarised in the next sections.

5.2.1.1 The XML Parser

The main goal of XML parsers is to help applications extract XML data from its plain textual format. An XML parser treats an XML document as a tree (see Chapter 2, a tree-structure representation of XML) which consists of a finite set of nodes and edges, the textual content is stored in leaf nodes. The role of this process in an holistic model is that it labels elements of the underlying XML data using an existing labelling scheme (see Chapter 3) and partitions the underlying XML data into a set of streams which contain the generated labels. There are two models of XML parsing: SAX and DOM. Since XML parsing is an essential precursor to any processing of an XML document, the XML parsing process is considered as the bottleneck to performance when processing XML data. It was hypothesized that a SAX parser shows superior performance when a serial access is required while a DOM parser supports random access of an XML tree nodes efficiently [169].

The implementation of the *holistic model* for processing XML queries in this thesis adopts SAX rather than DOM as the XML parser. As was mentioned in Chapter 2, a SAX parser can provide a simpler,high-performance, low-level access of XML document trees, whilst DOM parsers consume memory space and it may limit the size of XML documents processed in this model, alternatively it provides more functions. The main reason for selecting SAX as an XML parser for the *holistic model* implemented in this thesis was that the system needs to partition the XML tree nodes into inverted indexing lists according to their tag names (i.e., adopting *tag streaming scheme*). Consequently, a SAX parser leaves the choice of the underlying data structure up to the system, in other words, It allows applications to construct customisable data structures using a series of predefined parsing events (e.g., start and end of elements) [36].

In addition, the *holistic model* needs an efficient way to retrieve the logical structure information of the XML document trees in order to analyse the input XML documents as fast as possible using a little memory. More importantly, the aim of this study is to propose alternative approaches for processing XML twig pattern queries over indexed XML documents regardless their size. Thus, a SAX parser can parse XML document

trees of any size (e.g. number of XML nodes or disk-size) even if they are larger than the available resources [194]. The main memory requirement of a SAX parser is proportional to the longest depth of the parsed XML tree and the maximum amount of data involved within a single element [36]. Lastly, the implementation of a SAX parser is freely available in several platforms, such as Java [36] and C++ [183]. These are important factors in choosing a SAX parser in this thesis as the XML parser.

To conclude this section, the process of a SAX parser can be seen as *depth-first traversal* of the XML document. During the traversing process, a SAX parser starts at the root node, moving downward to the left-most, deepest leaf nodes and upward then back to navigate the rest of XML tree branches until the whole XML tree has been covered. When each element is encountered, a set of hierarchical relationships can be recognised. The next section describes the use of SAX parser to traverse and label the XML tree nodes in the *holistic model* used in this study. It also discusses the algorithm of parsing and labelling XML tree nodes.

5.2.1.2 Node Labelling Scheme

A labelling scheme provides efficient access to XML tree nodes by supporting the navigational operations and the evaluation of the main axes for XML query languages [94]. In particular, most query processing algorithms [40, 5, 19, 22, 89, 185] on XML documents rely on labelling schemes, such as region-based [253, 90] and prefix-based labelling schemes [76, 147]. The labelling scheme associates each XML element with an unique label (based on a labelling scheme) which records its positional information within an XML tree. Range-based labelling scheme which is commonly used in XML query processing algorithms [40, 144, 145, 89, 185, 53, 132, 70]. In this labelling scheme, each element is assigned with a triplet as $(start, end, level)$. *Start* and *end* records values of positions corresponding to the opening tag $< tag-name >$ and the closing tag $< /tag-name >$. *Level* represents the depth of an element within an XML tree. Therefore, the hierarchical relationships can be determined efficiently. An example of this labelling scheme is depicted by Figure 5.2a, each element is labelled with rang-based encoding scheme, for instance, *e-node* which is located in the first level and is associated with 1 and 15 values to represent start and end parameters, respectively. Labelling schemes were further discussed in Chapter 3.

All holistic twig matching algorithms which are described in this thesis are based on a range-based labelling scheme. Thus, the selected labelling scheme to label XML tree nodes in the *holistic model* is range-based labelling scheme [90]. One advantage of this labelling scheme is that it can be produced in a single scan of the original XML document and is less complicated than the other approaches [89, 98]. The main weakness of this approach is that it is not a dynamic labelling scheme, in other words, the XML document must be relabelled when an update occurs. As was mentioned in the previous section, a SAX parser invokes a set of callback methods to inform the subsequent operations (e.g.

XML labelling, data partitioning and tag indexing) the XML document structure (e.g. names,attributes and nested elements). These callback functions are: *startDocument()*, *endDocument()*, *startElement()* and *endElement()*. The former two functions are responsible for constructing auxiliary data structures and operations at the start and end of an XML document, respectively. The latter two are functions can be used to perform operations when the opening and closing tag of an element are encountered, respectively. There is a SAX parsing event which is responsible for representing the textual content of an XML element. This event can be captured through predefined function called *characters()* in the Java implementation of SAX [36].

The process of XML parser and node labelling is given in Algorithm 1. The whole process is committed in two of SAX event handlers, namely *startElement()* and *endElement()*. The events are passed to the code through a SAX listener (line 7), if the SAX listener encounters an opening tag of an element, it triggers *startElement()* (lines 15-33) which records the start position of the current element when traversing the XML document in depth-first scanning. The level information is recorded as well. The current information is stored in two global variables *orderStack* and *level* to be added later to the end position value in order to constitute the range label of the current element. Another global variable is *childStack* which keeps tracking of the number of children of the current element to ensure the containment because in region-based labelling scheme, the end values of an element must be larger than all the start and end values of its immediate children and its descendants. This is handled when the algorithm encounters an closing tag of an XML element. Meanwhile, the attributes of the current processing element are passed through the SAX's event as a set of attributes because of the existence of attributes within the opening tag of an element. It is worth noting that most existing XML query processing algorithms [40, 147] treat an XML attribute as an element or more precisely a leaf element because it can not contain another element as well this thesis does. The algorithm iterates over the attributes of the current undergoing element and assigns each attribute its unique region label (lines 26-33). Additionally, *dataPartition()* function is called to insert the label of the current attribute to its corresponding partition (see Algorithm 2). The methodology for data partitioning of an XML document is discussed in the following section. When an closing tag of an element is encountered, *endElement()* is invoked (lines 35-43). This function is used to construct the entire region label of the current element. If the current element has at least one child node, its end value is set by the increment of the current order, otherwise it is set to be the current order. The information of the current element label is sent to *dataPartition()* for storage in the corresponding partition (line 42-45). The other SAX's events are omitted because they do not contribute to the process of labelling XML tree nodes in which the logical structure of the XML document is the main concern.

An XML query is composed of structural search and content search [235]. The content search is used to filter the structural solutions based on the specified value comparison [236]. In the context of XML, a value, which is each non-tag text, is contained with the opening

and closing tag of an element and it must be a leaf node in the tree representation of the XML document. The parent of each value node (a.k.a text node) is known as the property of this value. During the labelling process, a value can be obtained through a SAX's handler *characters()* and is processed prior to *endElement()*. The result of *characters()* is an array of characters which contains the textual content of the current processing element. There are two times when a SAX event encounters a text node in the XML tree. Firstly, scanning the content values of the XML attributes (line 31). Secondly, parsing the textual content of an XML element after the completion of a *startElement()* event and prior to the process of an *endElement()* event. In the *endElement()* function, if the current element is associated with textual data, the information of its label and the textual value are passed to the partitioning algorithm to construct the tag-based streaming list for each distinctive tag in an XML document (line 38-42). Nevertheless, there are several approaches to facilitate the process of XML queries which contain value-based constraints [235, 236]. These approaches are based on the idea of utilising the relational schema in relational databases to store values of XML nodes, the content-search then can be performed prior to the structural search in order to speed up the performance. The research work proposed in this thesis can be combined with these approaches since they used *TwigStack* algorithm to perform the structural search. However, the combination of approaches proposed, in this research work, is beyond the scope of the thesis.

As was reported in Chapter 4, the main weakness of the existing labelling schemes, which adopted in the previous holistic twig matching algorithms, may be tackled by increasing the meta-data of the existing labelling schemes. The study of this thesis aims to improve the process of XML queries in holistic model. Therefore, this thesis proposes a new approach to be incorporated into the existing region labelling scheme in order to accelerate the determination of a Parent-Child relationship in this model. The fundamental idea of the new approach is to augment the existing region labelling scheme with an extra parameter using the property of prime numbers. The new technique will be further discussed in Chapter 6.

To sum up, the *holistic model* adopts a SAX parser to map the underlying XML data into a set of streaming lists which contain the labels of the parsed XML tree nodes. As the performance of a SAX parser was discussed in the previous section, the performance of a region encoding labelling scheme is determined by the performance of the XML parser which is O(n), where n is the total number of XML tree nodes [36, 90]. Similarly, the memory requirement for region encoding labelling scheme is proportional to the longest path in the input XML tree and the destiny of the textual content of XML tree nodes [3]. The system uses a single SAX parser process in order to construct the components of the storage model (see Figure 5.1). An XML document is parsed only once unless the XML document is updated or the system may require additional knowledge prior to commencing node labelling scheme. The following section describes the process of organising the

storage into *partitions*. Moreover, it explains the algorithm of partitioning the underlying XML tree nodes into a set of stream labelled with a region labelling scheme.

5.2.1.3 Data Partitioning Scheme

In the *holistic model*, the storage is organised into partitions which are a set of streams annotated with a labelling scheme. The result of this stage, in this thesis, is a set of streams labelled with a region encoding labelling scheme. One of the core segment of the *holistic model* is the resource model which is a persistent model and can use the file system as a simple storage engine (i.e., storage device see Figure 5.1). The resource model receives a request from the main component of the evaluation stage, namely the query processor. The main purpose of this phase is to pre-process XML documents to facilitate the underlying execution model. This process aims to minimise irrelevant data access in the structured retrieval of an XML document by returning always the smallest unit of a document answering the input query [54, 151].

The pioneering work of [40], proposed the first holistic twig matching algorithm in order to overcome the limitations of the binary structural joins algorithms which usually generate unnecessary intermediate results. A significant assumption of the original holistic twig matching algorithm is that an XML document tree is clustered into tag streams in which all elements with the same tags are grouped together and each element is associated with an interval encoding (e.g. range-based label) [54]. This clustering technique is known as *tag streaming* and the combination of XML structural index strategies with labelling schemes is called an *XML streaming scheme* (see Chapter 3). This model adopts the XML *tag streaming scheme* which uses an XML tag as the key to partition the entire tree.

The algorithm for partitioning an XML document tree into disjoint inverted lists which contain either labels of the XML elements or a pair of labels and values of the XML elements associated with text nodes is presented in Algorithm 2. All the information necessary to construct the inverted lists are passed to the algorithm as arguments. The ID of the XML document is used to create a new directory for the input document in order to store related partitions in the same directory which is, in turn, subdirectory of the main directory in the system (i.e., DB). The input information includes a tag name of the input XML element, the positional information of the context element and the textual content (if any exists).

Firstly, the algorithm checks whether the inverted list corresponding to the current element's tag has been created or not (lines 3-6). The *dataPartition* method (lines 6-9) simply appends the new label into its corresponding an inverted list and then sorts all the existing labels based on their start values in order to keep the inverted list in an order which conforms to the document order. The system stores each inverted list (i.e., tag streaming list) into a separate file on disk. The inverted lists are further indexed automatically by the file system, thus the relevant inverted lists to the input XML query nodes can be quickly accessed during query processing [39, 236]. If the current element is associated with a

Algorithm 1: Region Encoding Algorithm

Input: an XML file
Result: XML elements labelled with the original range-based labelling scheme

```
1  // initialization
2  IntegerStack orderStack = ∅ // stack holds the start values of the current processing
     elements.
3  IntegerStack childStack = ∅ // stack holds the number of children for each element.
4  order = 0 // the start value of the current element.
5  level = 1 // the recent accessed level.
6  docID = xml.name // the ID of the XML document.
7  event = saxParser(xml) //sax method returns a sequence of events.
8  while ¬eof(event) do
9      if event is open tag then
10         startElement(event.tag,event.attributes)
11     else
12         endElement(event.tag)
13     event = saxParser(xml) // get the next SAX event.
14 Procedure startElement(tag,attributes):
15     order = order + 1 // generate a sequential integer number.
16     push(orderStack,order)
17     level = level + 1
18     if ¬ isEmpty(childStack) then
19         x = pop(childStack)
20         x = x+1
21         push(childStack,x) // for the parent element
22         push(childStack,1) // for the current element
23     else
24         push(childStack,1)
25         // this is for the root node because it does not have a parent.
26     while ¬ isEmpty(attributes) do
27         order = order + 1
28         x = pop(childStack)
29         x = x+1
30         push(childStack,x)
31         call dataPartition(docID, attributes.current, order, order, level +
             1, attributes.current.value)
32         // see Algorithm 2
33         attributes = attributes.next
34
35 Procedure endElement(tag):
36     level ← level − 1
37     if pop(childStack) ≠ 1 then
38         order = order + 1
39     if characters().length > 0 then
40         call dataPartition(docID, tag, pop(orderStack), order, level, characters())
41         // see Algorithm 2
42     else
43         call dataPartition(docID, tag, pop(orderStack), order, level, empty)
44         // see Algorithm 2
```

text node (lines 10-17), the algorithm combines the text node with the label of its parent element into a new inverted list to store the textual content. Lines 8 and 16 ensure all the entries in the value inverted list are sorted based on the start values of their labels.

Algorithm 2: Tag Partitioning Algorithm

```
   Result: Store the label of the input element in the corresponding partition
1  Procedure dataPartition(docID,tag,start,end,level,text):
2      // Create an inverted index list contains labels of nodes sharing the same tag
         name equals to tag and values are combined to their parent elements' labels in
         separate inverted list.
3      if ¬exist(File(DB/DocID/tag)) then
4          newFile = createFile(DB/DocID/tag)
5      else
6          newFile = openFile(DB/DocID/tag)
7      // newFile:write( (start,end,level) )
8      // Sort all labels in newFile based on their start values.
9      newFile:close()
10     if text.length > 0 then
11         if ¬exist(File(DB/DocID/tag_value)) then
12             newFile = createFile(DB/DocID/tag_value)
13         else
14             newFile = openFile(DB/DocID/tag_value)
15         newFile:write( (start,end,level), text ) // Create an inverted list contains the
             label of the property node and its value.
16         // Sort all labels in newFile based on their start values.
17         newFile:close()
```

The main reason for the combination of element's value and label in a single inverted list is that the content search can be performed while scanning inverted lists, so that holistic twig matching algorithms can improve the efficiency of the structural search by reducing the size of inverted lists when processing XML queries with content constraints [236]. This approach also avoids labelling text nodes separately which leads to a reduced number of inverted lists in the system. For illustration, examples of two XML datasets, which are widely used in the literature and are rich with textual content, have been tested in terms of the number of labelled nodes and inverted lists in order to compare the two approaches for partitioning XML documents. Table 5.1 compares the breakdown of the real world DBLP dataset and the benchmarked XMark dataset according to the number of labelled nodes (i.e., elements, attributes and text nodes) and the number of inverted lists. It can be seen from the data in Table 5.1 that *E&T* approach (i.e., elements and text values are encoded together) has a superior performance in terms of space management to the original work introduced in [40]. Having discussed how to construct XML *tag streaming* lists for the underlying XML document, the next section describes the process of indexing tag names of XML elements in order to facilitate the determination of a Parent-Child relationship in TPQs. *Tag indexing* is a new indexing technique which is proposed in this thesis.

Table 5.1: Features of two examples of XML datasets in terms of the number of labelled elements and inverted lists. *E-T* stands for elements and text values are encoded separately, while *E&T* means text nodes are combined with their parent elements.

Dataset	# of labelled nodes			#s of inverted lists		
	E-T	*E&T*	saving	*E-T*	*E&T*	saving
DBLP (127 MB)	6,771,148	3,736,406	44.8%	388,630	79	99.9%
XMark (116 MB)	3,221,925	2,048,193	36.4 %	353,476	138	99.9%

5.2.1.4 Tag Indexing

The *tag indexing* is a simple indexing scheme which provides a mapping from a string to a prime number. This novel technique contains a collection of 2-tuple (tag name,unique prime number) and is implemented as *hash* table in order to facilitate the look-up of the corresponding prime number associated with a particular tag name during query processing, where a key is a tag name and a return value is an unique prime number. This is very important component to interpret the information encoding within internal, branching elements in an XML document. The construction and utilisation of this new approach will not be described here, as it will be discussed in Chapter 6. The following section describes the process of evaluating XML queries in the *holistic model* and its subsections discuss the roles of processes of the execution model.

5.2.2 Execution Model

The on-line phase of the *holistic model* performs the query processing phase of the designed system. It is sequentially organised in two fundamental stages: *Query Constructor* (a.k.a query parsing or query analyser) and *Query Processor* (see Figure 5.1). Traditionally the first step in any XML query processing system is to load and parse the input XML query. The purpose of this step is to translate the input XML query into an optimised evaluation query plan (i.e., twig pattern query). The next conventional step is to evaluate the parsed XML query using a holistic twig matching algorithm. The structure and functions of each stage will be explained in the following sections.

5.2.2.1 Query Constructor

Several query languages have been proposed in order to retrieve information within an XML document such as XPath [222] and XQuery [224]. XPath is the standard language for traversing paths in an XML document. Generally, an XML query can be decomposed into a set of *path expressions* which is the basic building block of XML query languages. The query constructor compiles the input XML query into , twig pattern query, a logical query plan which can be used later by a physical query plan implemented in a query processor. This is due to the fact that TPQ is the most simple model and has been adopted as the logical query plan in many approaches [40, 147, 89, 87, 19, 21]. The query constructor is mainly responsible for transforming the input XML query into its twig representation in

order to facilitate the query evaluation process. Firstly, an XML query is expressed in an XPath-like format which is a fragment of *XPath 2.0* specification [222]. Having formulated queries, the system then builds the corresponding twig pattern query (TPQ) of the path expression.

This thesis specifies a grammar for an XML query, using the basic EBNF notation [178]. The grammar of XML query expressions addressed in this thesis is presented in Figure 5.4. An XML query is accepted as a string in the system, to better demonstrate the accepted format of an XML query see Example 5.1. It should be noted that all XML query languages request an XML user to specify the exact XML document whose information shall be retrieved. As a result, the system receives an XML query in the form of a pair as (XML query expression, Document name). In addition, the role of the query constructor can be extended to classify the input XML queries according to their query types because a query type can determine how a query can be evaluated by the query processor. It can be seen from the introduced grammar of *path expressions* that the designed system supports the evaluation of a wide range of query types including the main axes of XPath [222], sequence operator and positional constraint. The query classification is an important step because the XML query processor applies different types of holistic twig matching algorithms in order to improve the efficiency, scalability and performance of the query evaluation plan for a certain query type. The result of this phase yields an optimised twig pattern query based on the *pattern expressions* addressed in this thesis (see Figure 5.4). The process of extending the generated TP to capture semantics of ordered axes and definitions of TPQs for different classes of XML queries were discussed in Chapter 4.

Example 5.1. *Consider the XML tree t_1 of Figure 5.2a, suppose this XML document is parsed, indexed and partitioned in the system. This query is looking for a x-node whose content is "is x" which has a parent a-node and a sibling y-node with a child f-node. This XML query can be expressed based on the grammar introduced in this thesis as //a[/x="is x"]/y/f. The corresponding TPQ representation of this XML query is depicted in Figure 5.5, where the nodes stand for terminal/non-terminal symbols of the introduced grammar. Consequently, the system accepts this query in the form* $Q = (//a[/x = \text{"is x"}]/y/f, t_1)$

$TPQ \rightarrow Pattern$
$Pattern \rightarrow Step(Predicate)^* \ (Pattern)?$
$Step \rightarrow Axis\ NodeTest$
$Predicate \rightarrow [\ Pattern \mid OrderSpec \mid Digits \mid GeneralComp\ \text{“}\ ,\ string\ ,\text{”}\ ]$
$Axis \rightarrow / \mid // \mid /following:: \mid /following-sibling:: \mid /preceding:: \mid /preceding-sibling::$
$OrderSpec \rightarrow NodeTest\ NodeComp\ NodeTest \mid Pattern\ NodeComp\ Pattern$
$NodeComp \rightarrow << \mid >>$
$Digits \rightarrow [0-9]^+$
$GeneralComp \rightarrow = \mid != \mid < \mid <= \mid > \mid >=$
$NodeTest \rightarrow String$

Figure 5.4: A grammar of XML query expression used in this *holistic model*.

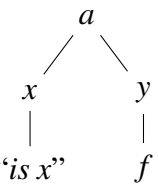


Figure 5.5: A twig representation of an XML query in example 5.1.

5.2.2.2 Query Processor

An XML query is normally processed in two steps: query constructor and query processor. The former is responsible for constructing an optimised query plan for evaluating the input query. A query processor then interprets the plan. The query is then evaluated by the appropriate holistic twig matching algorithm (i.e., a physical query plan) for rendering a results set. The query processor consists of groups of holistic twig matching algorithms to process different classes of XML queries over indexed XML documents. The query processor can evaluate the following query types (see Chapter 2): predicate (branching), non-predicate (simple path), predicate with order constraints, predicate with positional constraint and non-predicate with positional constraint. In addition, the query processor can evaluate XML queries in two different manners either *top-down (i.e., two-phase)* or *bottom-up (i.e., one-phase)*. The following subsections describe the core algorithms in each category. The functionality and coverage for each group of algorithms will be described briefly as well.

5.2.2.3 Basic Two-Phase Approach

In the basic two-phase category, both predicate and non-predicate XML queries can be processed in *TwigStack* [40] manner. A structural search is processed in two phases. In the first phase, the input XML query is decomposed into a set of individual single paths and the structural join operations are performed using a chain of stacks. The second phase is responsible for merging join the intermediate results of the first phase in order to eliminate useless single paths. In this *holistic model*, three two-phase twig matching algorithms are implemented from scratch. Two are previously proposed in the literature, namely *TwigStack* and *TwigStackList*, respectively in [40] and [144]. The third one is originally proposed in this thesis and it can be seen as an extension of *TwigStack* to process Parent-Child axes efficiently. The name of the new algorithm, *TwigStackPrime* is driven from the combination of the original *TwigStack* with a new technique which is based on *prime numbers*. It utilises *tag indexing* in order to facilitate the query evaluation. *TwigStackPrime* will be further discussed in full detail in Chapter 6.

5.2.2.4 Ordered Two-Phase Approach

The work of this thesis aims to improve the efficiency of holistic twig matching algorithms and increase their query coverage. Up to now, there is no holistic twig matching algorithm

[145, 118, 173] which can consider the semantic of order axes and sequence operators introduced in the XPath specification [222]. As was discussed in Chapter 4, this thesis incorporates the semantics of order constraints into TPQs in order to support order XML queries. As a result, novel set of ordered holistic twig matching algorithms have been proposed which are either an extension of *TwigStackPrime* as *OTJPrime* or a combination of *TwigStackList* and *TwigStackPrime* as *OTJPrimeList* (abbreviated as OTJPL). Moreover, *OTJPL* algorithm was further optimised to process *following-sibling* and *preceding-sibling* efficiently. The refined version of *OTJPL* uses a modification of level split vector which was proposed in [89] so that the new algorithm is called *OTJPrimeMultiList* (OTJPMultiL, for short) which stands for Ordered Twig Join Prime Multi List. The new approaches will be described later in Chapter 7.

5.2.2.5 Basic One-Phase Approach

In the context of XML query processing, reviewing the state of the art of XML query processing algorithms leads to the fact that most of the sophisticated holistic algorithms fall in this category. The one-phase approach alleviates the process of structural search in the second phase of *TwigStack*-based algorithms by eliminating extensive merge join operations in order to join structurally branching nodes of TPQs. A combination of preorder filter strategy in the one-phase algorithm with the postorder filtering strategy in the one-phase has been proven to be problematic [185, 132, 89, 22, 20]. Therefore, a set of one-phase holistic algorithms were proposed, in this thesis, in order to improve the efficiency, scalability and performance of one-phase holistic twig matching algorithms. The basic notion is to combine the the efficient filtering of useless elements introduced in *TwigStackPrime* with the efficient data structure for storing intermediate result introduced in the previous work [132, 89]. *TwigPrime* algorithm and its refined versions were proposed in this thesis to demonstrate the superiority of the new approaches to the previous one-phase holistic algorithms. Further explanation of the new approaches will be given in Chapter 8.

5.2.2.6 Ordered One-Phase Approach

Similar to the approaches proposed in the previous section, a group of order one-phase holistic algorithms were proposed based on the efficient selection of useful elements introduced by *OTJPrime*, *OTJPL* and *OTJPMultiL* in section 5.2.2.4. This combination enables the previous mentioned one-phase holistic algorithms to process TPQs which contain order constraints and sequence operators efficiently without performing post-processing steps. A more detailed account of techniques to combine different approaches in order to support forward and backward order axes as well as sequence operators in one-phase holistic algorithms will be given in Chapter 9.

5.2.2.7 Positional One-Phase Approach

XML query languages such as XPath [222] and XQuery [224] supports functions beside the basic thirteen axes introduced in the specification of XPath (see Chapter 4). Positional predicate or function is commonly used to increase meta-data of TPQs as well as add a restricting condition. Previous studies of XML query processing have not supported positional predicates in an advanced preorder filtering strategy [217, 70]. Furthermore, the increasing size of XML documents and the complexity of evaluating TPQs with positional predicate pose a challenge performance to the existing positional XML query algorithms. Therefore, this thesis proposes a novel positional one-phase holistic algorithm which can filter out several useless elements based on a combination of novel techniques. Chapter 9 describes the proposal of a novel positional holistic twig matching algorithm.

5.3 The Implementation of the Experimental Framework and Testing Platform

The holistic-based system for XML query processing was implemented using Java programming language (with JDK 1.8) and Eclipse IDE. As was described in Section 5.2, the system consists of two stages to facilitate the query evaluation process. The next two sections describe the implementation of these stages.

5.3.1 The Storage Model

As was discussed in Section 5.2.1.1, the system used a SAX parser to extract the structure and content of an XML document. The Java implementation of a SAX parser was adapted to generate SAX events for the construction of the storage model. By scanning the XML document through a SAX parser, each node of an XML tree is assigned a label as was described in Algorithm 1. During the labelling process, the textual content involved within a text node is stored alongside the label of its parent in a text file (i.e., inverted list). When a label of an XML node is generated, the label is stored in a text file based on the node's tag name. The text files are sorted in ascending order of their start values. The process of partitioning and sorting text files is coded as described in Algorithm 2.

5.3.2 The Execution Model

The implementation of the execution model is composed of two stages. The first stage implements a query analyser to tokenise a given query into a set of context-nodes in order to build tree-like representation of the query (i.e., TPQ). Then, the query is further tested to check whether its nodes correspond to tag names of the XML document or not. In addition to checking the validity of the query on the given XML document, the type of the query is classified according to the class of query supported in this model as was

described in Section 5.2.2.1. Having identified the type of the query, the system requests the necessary information to evaluate the query from the storage model and invokes the appropriate holistic twig matching algorithm based on the query type. The second stage is the implementation of groups of holistic twig matching algorithms which are based on inverted lists and can evaluate the query in a single forward scan of the lists.

5.3.3 Platform Setup

All the algorithms implemented in the system were written in Java with JDK 1.8. In this study, a set of experiments were conducted to assess the characteristics of the proposed holistic algorithms and test the efficiency, scalability and performance of holistic twig matching algorithms in order to compare the new holistic algorithms with the existing ones. All the experiments were performed on a laptop with 2.9 GHz Intel Core i5 and a RAM memory of 8 GB 1867 MHz DDR3. 500 GB HD Size with Mac OS Extended (journaled) format. This laptop runs a Mac OS X El Capitan as the operating system.

5.3.4 Testing the Holistic Model

The *holistic model* system was tested practically using different levels of functional testing for a software: unit testing, integration testing and system testing. The goal of unit testing here is to test individual components of the system to ensure that they function properly and produce the required data. In the designed system, a variety of XML datasets with different characteristics (see Section 5.4) were used to identify how the node labelling process is affected by the properties of XML documents in order to generate XML labels effectively. Also, complex twig queries were tested in order to examine the building of their twig representations in the query constructor. For the process of generating inverted lists, the result of unit testing should be that the number of lists generated equals the number of distinctive tags plus the number of text nodes in the benchmarked XML datasets.

Having identified the individual units work correctly, the reason for conducting integration testing is that some units, such as query constructor, query processor, tag indexing and resource model, are integrated as was illustrated in Figure 5.1. In this level of testing, the system checks how the optimised query model would pass correctly to the query processor and then the retrieved inverted lists are sufficient to evaluate the twig queries. Since the system is implemented from scratch, the system was eventually tested as a whole to verify that it meets the main functional specification which is the evaluation of XML queries on indexed XML documents and renders query result sets for testing queries. This is done by comparing the functionality of the designed system with state of the art XML query engine, namely Saxon 9.7 [160]. The outcomes of the system testing should be that the completeness and the correctness for the *holistic model* system are demonstrated with respect to a specification of XML queries addressed in this research. In the other way, the designed system supports all the queries and all supported queries give the correct answers.

Therefore, the system can be used as the test-bed on which the efficiency, scalability and performance of holistic twig matching algorithms can be assessed and the hypothesis, which was introduced in this thesis (see Section 4.3.2), can be tested as well.

The next section discusses properties of the available XML datasets which are widely used to test the performance of XML techniques.

5.4 An Overview of XML Datasets

This section gives an overview of the general classification of XML datasets. It describes the existing XML datasets used in the literature for XML testing and benchmarking. XML datasets may be divided into three sub-groups: real-world, benchmark and synthetic datasets. A summary of the properties of each dataset will be discussed further for real-world XML datasets in sections 5.4.1 and benchmark datasets in Section 5.4.2. In addition, two synthetic XML datasets were generated in order to cover testable aspects of performance which may be missed by the existing real-world and benchmark datasets. Section 5.4.3 provides a brief review of the most popular synthetic datasets which are widely used in the context of XML query processing.

5.4.1 Real-World Datasets

A real-world XML dataset is a single XML document which contains real data. The existing XML documents differ in three aspects: the depth, the breadth and the number of XML nodes to reflect the variety of XML documents. In addition, XML documents can be either highly structured (*data-centric*) or irregularly structured (*text-centric*). The characteristics of an XML document may affect the performance of query evaluation process [22]. Some of these datasets were used in the experiments for testing the efficiency, scalability and performance of holistic twig matching algorithms previously proposed in the literature (see Chapter 3). The subsequent sections describe the properties of the most commonly used real-world datasets.

5.4.1.1 DBLP Dataset

DBLP is an acronym of Digital Bibliography and Library Project. It is a huge XML dataset which contains bibliographic information on major computer science publications such as the VLDB Journal and ACL conference. The original version of DBLP has indexed more than three million publications and the XML file size is up to 1.77 GB and it can be obtained from [32]. The university of Washington XML repository provides a small version of this dataset for the purpose of the experimental evaluations in the XML research community [162]. This real-world dataset is widely used in the experiment evaluation of previous research studies as in [5, 9, 7, 154, 114, 70, 245, 244, 118, 147, 237, 112, 53,

105, 253, 89, 146, 115]. The structure of this dataset is simple, wide and highly regular. The main characteristic of the small version will be summarised in Table 5.2.

5.4.1.2 TreeBank Dataset

TreeBank was designed by the Computer and Information Science Department at the University of Pennsylvania [162]. It is a single XML file of parsed English sentences tagged with parts of speech. The English Sentences were selected from a collection of three years of Wall Street Journal (WSJ) stories for syntactic annotation. The dataset is partially encrypted with respect to text nodes for copyright protection. The structure of the dataset is not affect by the encryption process. Since it has a complex, deeply recursive and irregular structure, it is considered as an interesting case for relative performance evaluations [185, 70, 89, 144, 147, 146]. The dataset contains a large number of nested structures (338,749) [19]. The XML file of TreeBank is a quite large in size with 82 MB and can be freely downloaded from the XML Data Repository [162]. Table 5.2 summarises features of this dataset.

5.4.1.3 Protein Sequence Dataset

Protein Sequence Dataset is published by Georgetown Protein Information Resource [44], and serves as a resource of integrated bioinformatics which records functionality annotated protein sequences. The structure of this dataset is considered to be simple and regular similar to DBLP [22, 66]. The XML file is about 683 MB and it is the largest dataset available in [162]. It is commonly used to experimental evaluations on the XML storage and the XML streaming techniques [57, 94, 239]. Table 5.2 presents some features of this dataset.

5.4.1.4 NASA Dataset

The NASA dataset contains an astronomical data which are converted from s flat-file format into XML by the GSFC/NASA XML Project. The XML file is 23 MB in size [162] and it has a shallow structure with a few recursive elements. Unlike TreeBank, the number of unique tags is small 69 distinctive tags comparing to 251 in TreeBank. This dataset has been used to test different aspects of XML application for processing XML queries [235, 18, 105], labelling XML elements [140] and compressing [251]. Some properties of the dataset will be given in Table 5.2.

5.4.1.5 SwissProt Dataset

SwissProt is a curated protein sequence dataset which describes the DNA sequences marked up in XML. The size of the XML file is 109 MB. The dataset is tightly structured and can be integrated with other datasets [162]. It is used in the evaluation of a variety of

XML technologies such as XML query processing [18, 188, 105, 149]. Table 5.2 illustrates some features of SwissPort dataset.

5.4.1.6 SIGMOD Record Dataset

SIGMOD Record dataset is the XML version of a portion of the ACM SIGMOD website which contains issues of SIGMOD Record. It is a tiny dataset since the XML file size is around 0.5 MB [162]. It is used to the relative performance evaluation of techniques on small XML databases [134]. The characteristics of the XML file will be presented in Table 5.2.

5.4.1.7 Mondial Dataset

Mondial is an XML file serves as a geographical database of diverse sources such as the CIA World Factbook, the International Atlas, and the TERRA database [162]. The XML file is small in size around 1 MB. It was considered in various experiments to test the performance of XML technologies [94, 96]. Table 5.2 provides some features of this dataset.

Table 5.2: Characteristics of the existing real-world XML documents.

Dataset	Size MB	# of nodes		Depth		Distinctive tags
		# of elements	# of attributes	Max	Avg	
DBLP	127	3332130	404276	6	2.9	40
TreeBank	86	2437666	1	36	7.8	251
Protein Sequence	86	21305818	1290647	7	5.15	68
NASA	23	476646	56317	8	5.58	69
SwissProt	109	2977031	2189859	5	3.55	99
SIGMOD Record	≈ 0.5	11526	3737	6	5.14	12
Mondial	1	22423	47423	5	3.59	55

Table 5.3: Characteristics of the existing Benchmarked XML datasets.

Benchmark	Key parameters	Dataset		# of users	# of queries
		# of files	Size	Max/Avg depth	
XMark	size factor	single	any size	12/5.5	single	20
XOO7	depth, fanout and amount of textual data	single	three fixed size: small, medium and large	7/5	1	23
TPoX	size and # of users	multiple	each file has size from 2 to 25 KB	template-based	multiple	17
XBench	size	single/multiple	four classes ranging from small with 10 MB to huge with 10 GB	limited	single	20
XMach-1	# of documents, elements, words in a sentence, probability of phrases and links	multiple	from 2 to 100 KB per file	6/3	multiple	11
The Michigan	size	single	single file with 739K nodes as default and can be scaled down and up to 10 times its original size	16/5	single	31

5.4.2 Benchmark Datasets

This section gives an overview of the most popular XML benchmarks. Generally, XML benchmarks are used for evaluating the performance of XML storage and XML query processors [195]. XML benchmarks may be classified into micro-benchmarks and application-benchmarks. Application benchmarks, such as XMach-1 [33], focus on assessing the performance of the entire system (i.e., XML databases) whereas micro-benchmarks, such as XMark [195], concentrate on XML processing operations [170]. The following subsections highlights well-know XML benchmarks which are commonly used in the literature.

5.4.2.1 XMark Benchmark

XMark Benchmark was developed as research work of [195]. XMark is a code generated dataset in a single XML file which simulates an Internet auction website, and can be seen as *data-centric* XML document. The XMark dataset generator is available on the XMark project website [195]. The dataset can be produced in any size based on predefined scaling factors. Regardless of the size of the dataset, the structure of the XML file has a considerable number of recursion and the maximum depth is fixed at twelve. It provides a twenty XML queries to test different aspects of XML technologies. Equally important, the authors of [78] proposed an XPath benchmark for the XMark document base. An XPathMark, which consists of a set of queries, was developed to evaluate tools which support the language XPath 1.0 and cover main aspects of the language including different axes, node tests, Boolean operators, references, and functions [87, 176, 89]. Therefore, it is the most commonly used XML benchmark to demonstrate the performance of variety of XML applications [144, 145, 87, 18, 185, 235, 52, 70, 140, 116, 105, 50]. The characteristics of the dataset generated by this benchmark are summarised in Table 5.3.

5.4.2.2 XOO7 Benchmark

XOO7 [35] can be considered as the XML version of OO7-Benchmark [41]. The data and queries of the original OO7-Benchmark were modified to test the characteristics of XML technologies. The XOO7 benchmark can generate an XML dataset in three different sizes: small, medium and large. Using predefined size of documents restricts the use of this benchmark for testing scalability. Similar to XMark, the depth is always five regardless the size of the XML file generated. Moreover, it provides twenty-three queries which cover search operations. Some features of the XML file is presented in Table 5.3.

5.4.2.3 TPoX Benchmark

TPoX is an acronym for Transaction Processing over XML developed by IBM [170]. It is a commercial application-benchmark to evaluate the overall performance of XML databases including storage, indexing and concurrency control. The dataset simulates a financial

multi-users system marked up in XML format which conforms to real-world Financial Information eXchange Markup Language (FIXML) schema. The dataset can be scaled from very small to extra large by identifying its depth and breadth (i.e., fanout). Seventeen queries have been composed in XQuery to cover fundamental operations. Table 5.3 shows a statistical information of this dataset.

5.4.2.4 XBench Benchmark

XBench is a family of XML benchmarks, and can be seen as template-based benchmark [247]. Because of the nature of XML as flexible data format which can be exploited to populate a document in different ways, an XML document can be further grouped into a single-document and multi-document. XBench can generate *data-centric* (DC) and *text-centric* dataset (TC). The dataset then can be in either a single-document (SD) or a multi-document (MD). Depending on the type and group of dataset, the *toXgen* tool is provided to generate four classes of datasets: DC/SD, DC/MD, TC/SD and TC/MD. The size of dataset can be in four different classes: small (10 MB), normal (100 MB), large (1G) and huge (10 GB). XBench provides twenty XML queries for evaluating search operations. The properties of the XML file are provided in Table 5.3.

5.4.2.5 XMach-1 Benchmark

XMach-1 (**X**ML Data **Ma**nagement Ben**ch**mark, version 1) is a scalable multi-user benchmark which was developed at the University of Leipzig, Germany [33]. It was the first XML benchmark proposed in the literature for evaluating the performance of XML data management systems. XMach-1 is composed of four parts: application server, XML database, loaders and browser clients. The size of XML file ranges from 2 KB and 100 KB. XMach-1 provides eleven queries to assess the performance of search and update processes.

5.4.2.6 The Michigan Benchmark

The Michigan Benchmark was designed by the authors of [192] at University of Michigan. It is a micro-benchmark for demonstrating the performance of XML processing engines. The dataset is generated in a single, default XML file which encompasses 739×10^3 nodes at fixed depth set to sixteen. The benchmark dataset can be scaled easily by increasing the number of nodes or fanout and the maximum size of dataset can contain 100 times the default number of nodes. In addition, the query-set is provided with twenty eight XML queries which cover all the aspects introduced in XMark benchmark [195], and three additionally queries concentrate on update operations. Table 5.3 presents the summary statistics for this benchmarked dataset.

Table 5.4: Statistical information about Random dataset used in this thesis.

Label	# of nodes (≈ thousands)
a	657
b	658
c	658
d	658
e	658
f	657
Summary	
Total # of internal nodes	1183298
Total # of leaf nodes	2764789
Total # of nodes	3948087
Max/Avg depth	13/7

5.4.3 Synthetic Datasets

Synthetic XML datasets are used to control the structure and join characteristics of the XML data so that the overall performance of holistic twig matching algorithms can be evaluated [112]. They are widely utilised in the relative performance evaluations for XML query processing methods [253, 40, 89, 22, 146, 38, 112, 101]. A synthetic Dataset is a single XML file which is generated using three parameters: *depth*, *fan-out* and *number of unique labels*. Synthetic datasets may be classified depending on the way the node labels are distributed into Random and *Zipf*. The next subsections describe the main features of each class and how they are utilised in this thesis.

5.4.3.1 Random Dataset

A Random dataset is created by using an XML data generator to generate the structural part of the XML tree (i.e., without text nodes) and the node labels in XML trees are uniformly distributed. In order to evaluate new querying approaches, this thesis creates a new Random dataset using the above mentioned parameters similar to the one produced by [40, 144]. The depth of data tree has a range from 2 to 13. The fan-out of nodes in XML trees is varied from 0 to 6. This dataset has six different labels, namely: *a, b, c, d, e, f* and the node labels were uniformly distributed. The XML dataset generated takes roughly 24.5 MB and contains about four million element nodes. In order to test the scalability of holistic twig matching algorithms concerning the number of nodes and the structural complexity of XML documents, this dataset can be scaled either down or up by using document size (i.e., the number of nodes) as a key to provide a scalable XML document. Since it is crucial for twig matching algorithms to be scalable, this study generated ten XML documents of increasing sizes with a scaling range from 3.1 MB to 31 MB. The document series is the following (sizes in MB): (3.1, 6.2, 9.3, 12.4, 15.5, 18.6, 21.7, 24.8, 27.9, 31). The characteristics of XML datasets which are randomly generated and used in this thesis are further illustrated in Table 5.4.

Table 5.5: Statistical information about Zipf dataset used in this thesis.

Label	% of nodes	# of nodes (≈ thousands)
a	38.55	1403
b	19.27	701
c	12.88	469
d	9.59	349
e	7.75	282
f	6.39	232
g	5.45	198
Summary		
Total # of internal nodes		1820881
Total # of leaf nodes		1820895
Total # of nodes		3641776
Max/Avg depth		26/18.5

5.4.3.2 Zipf Dataset

Zipf's law is a commonly used model to demonstrate how terms are distributed across a collection [151]. It states that the most frequent term will occur approximately twice as many times as the second most frequent term, three times as many as the third most frequent term, etc. A Zipf dataset was generated using the Zipfian distribution to spread the node labels within the dataset. In this research work, the Zipf dataset contains seven different labels from *a* to *g*, where *a* has the highest occurrences ($\approx 38.55\%$) and *g* has the lowest occurrence (($\approx 0.055\%$). Table 5.5 shows an overview of the distribution of node labels in the Zipf dataset. The fan-out has a maximum value of 2 and the depth of the dataset is up to 26. The XML file produced, with 25.5 MB, consists of 3.64 million nodes. The structure of the XML document has many recursions in some of its element names (i.e., recursive element names in a document path) and approximately equal numbers of internal and leaf nodes. The main properties of this dataset will be given in Table 5.5. The Zipf dataset generated in this work is similar to Zipf dataset used in [89, 22]

5.4.4 The Experimental Datasets

In evaluating the performance, scalability and efficiency of holistic twig matching algorithms, a set of XML datasets were used and this set was carefully selected from those that have been discussed in the previous section, real-world datasets in Section 5.4.1, benchmarked documents in Section 5.4.2 and synthetic XML files in Section 5.4.3. In order to test different aspects of XML irregularity, at least one dataset from each class was chosen to meet the experiments' objectives (see Chapters 6, 7, 8 and 9). From the real-world datasets, DBLP (tightly structured, see Section 5.4.1.1) and TreeBank (highly irregular, see Section 5.4.1.2) were included in the experiments to test two extremes of the spectrum with respect to the structural complexity. These datasets have been extensively used in the relative performance evaluations on XML query search [146, 40, 144, 89, 22, 185, 132, 181]. XMark

was selected over the other benchmarks because it is the most popular benchmarked dataset used to evaluate variety of XML techniques focusing on search functions [124, 181, 196] and was used in the experimental evaluations of the comparable XML twig matching algorithms [146, 40, 144, 89, 22, 185, 132, 181, 145]. Another reason for selecting XMark is that it provides a query-set consisting of twenty predefined queries which cover different aspects of XML query languages besides queries generated by XPathMark. Moreover, it is a single but scalable document which makes it appropriate for testing scalability. This study created five different XMark datasets with a scaling factor from 1 to 5.

In addition to the three datasets, synthetic datasets were used in the experiments in order to test as many aspects of irregularity in XML documents as possible. Random and Zipf datasets were adopted in the experiments as described in Sections 5.4.3.1 and 5.4.3.2. Datasets of this complexity in terms of structure are widely used to demonstrate suboptimal performance of holistic twig matching algorithms [89, 144, 40]. Furthermore, they were designed to test the performance of holistic algorithms where the XML combines features of DBLP and TreeBank being structured and deeply recursive at the same time. The Random dataset will be used to facilitate the process for evaluating the scalability of the new approaches using ten different document sizes. Table 5.6 presents all datasets used in the experiments of this thesis and their sizes.

Table 5.6: The experimental datasets and their sizes.

Dataset	Size in MB	Experiment
DBLP	127	Chapters 6 and 8
TreeBank	86	Chapters 6, 7, 8 and 9
XMark (default)	116.5	Chapters 6, 7, 8 and 9
XMark2	233.7	
XMark3	351.1	
XMark4	468.4	
XMark5	585.5	
Random (default)	24.5	Chapters 6, 7, 8 and 9
Random1	3.1	
Random2	6.2	
Random3	9.3	
Random4	12.4	
Random5	15.5	
Random6	18.6	
Random7	21.7	
Random8	24.8	
Random9	27.9	
Random10	31	
Zipf	25.5	Chapter 8

Generally, the relative performance evaluations in the context of XML query processing, such as [40, 89, 144, 132] and the one conducted in this study, are based on a set of XML queries which are executed on a set of XML datasets. In terms of query-set, a set of queries for each dataset will be described in the corresponding experiments. The testing queries

are classified into groups and each group represents a query type that is supported in this study (see Section 5.2.2.1). All queries tested in the evaluations are described in full detail in Chapters 6, 7, 8 and 9. The next section gives an overview of the statical procedures to analyse the experimental data in order to draw rigorous conclusions.

Table 5.7: Criteria for selecting statistical tests.

Type of analysis	Parametric	Nonparametric
Describe one group	Mean, SD	Median
Compare two unpaired groups	Welch's t test	Mann-Whitney U test
Compare two paired groups	paired t test	Wilcoxon test
Compare more than two independent groups	One-way ANOVA	Kruskal-Wallis test
Compare more than two dependent groups	Repeated-measures ANOVA	Friedman test

5.5 Data Analysis

This section gives an overview of the procedures involved in analysing the experimental data. This research study will present a set of experiments which aim to compare the relative performance of existing holistic twig matching algorithms for the purpose of evaluating the performance of new holistic twig matching algorithms based on *Child Prime Labels*. Since Computer Science is believed to be an experimental science [104], experimental analysis of the performance for a proposed method, comparing with a set of algorithms, is a crucial task in an investigation [82]. Statistical analysis is fundamental to all experiments which use statistics as a research methodology, such as the one conducted here [65, 63, 37, 100]. Having identified the design of research and types of collected data (see Chapter 4), statistical techniques will be used to analyse the data collected in order to decide whether to reject or accept the research hypothesis introduced in Section 4.3.2.

As was discussed in Chapter 4, testing the hypothesis requires comparisons of groups (i.e., holistic twig algorithms) in terms of outcomes (i.e., query response time and memory consumption) so that inferences can be drawn. Several factors determine what statistical procedures will be suited for testing the research hypothesis. These factors are the number of *independent* and *dependent* variables, the underlying measurement scales, such as a continuous variable (e.g., query processing time = 2 seconds) and a categorical variable (e.g., TwigStack = 1 and TwigStackPrime = 2), and normality distribution of scores (i.e., the Gaussian Population). Before proceeding to discuss the common statistical methods, the normal distribution is important because statistical tests are classified into two families based on their dependencies on parameters (i.e., mean and standard deviation) to describe the distribution of variables. Hence, the names are *parametric* and *nonparametric*. Tests referred to as *parametric* assume measurements come from a normal distribution, while *nonparametric* functions do not make assumption on the sample distribution [168, 82, 74]. Table 5.7 presents the most common statistical tests in computational experiments for the relative performance evaluations [104, 82, 37].

The statistical significance testing has a pivotal role in providing assessment as the observed relationships or differences reflect pattern than chance [63]. Thus, *Null Hypothesis Significant Testing (abbreviated as NHST)* is a statistical function for testing whether the factor has an effect on the observation or not. The main steps of NHST can be described as follows: developing the first statistical hypothesis (i.e., the null hypothesis denoted as H_0) which is generally assumed to be true until evidence indicates otherwise,

developing another statistical hypothesis, called alternative hypothesis and denoted as H_A which is the counterpart of the null hypothesis and finally running the appropriate statistical tests to determine *p-value*, which is a probabilistic abstraction indicating the probability of error involved in rejecting the null hypothesis [168], the null hypothesis can be rejected if the *p-value* is lower than 0.05 the conventional significant level. By way of illustration, the null hypothesis in this study will be formulated as "there is no significant difference in the performance of the algorithms compared ". The present study concerns both the time-related performance and the size-related performance for a set of holistic twig matching algorithms. Since the size-related performance can be evaluated using a simple data analysis in conjunction with graphical representation tools, but the time-related performance will be further tested for the statistical significance of the results. The Shapiro–Wilk test [74] was carried out to test the normality of query processing times of all query-sets in the experiments in order to choose the appropriate statistical tests from Table 5.7. Shapiro–Wilk test was chosen from the other normality tests because it yields the exact significance in most cases as reported in [74]. The outcomes turned out to reject the idea that the populations are normally distributed. Thus, the Mann-Whitney U test [165, 168] was used to compare two independent data samples which are , in this research, the results of running a holistic twig algorithm n times compared to another holistic twig algorithm in a pairwise comparison [37]. In the context of Computer Science, the most common rule is to have the number of runs $n = 30$ [104, 37]. However, since large numbers of runs can be carried out to evaluate properly the behaviour of holistic twig matching algorithms, this research has a number of runs $n = 100$ in the experiments conducted later in Chapters 6, 7, 8 and 9. The Kruskal-Wallis test [219] was also considered to compare more than two groups (i.e., a set of holistic twig matching algorithms) as an extension for the Mann-Whitney U test. It can be used for performing multiple comparisons between various algorithms by computing the differences in the performance based on the median [82, 219]. Specifically, for each testing query the Kruskal-Wallis test was run first to see whether there is a difference in the performance between two comparable algorithms at least. Then, the Mann-Whitney U test were carried out on each pair of groups to cover the total number of possible paired comparisons. As an illustration, for k groups, the total number of possible paired comparisons $= \frac{(k \times (k-1))}{2}$.

It should be noted that the *p-value* can not be used to assess the magnitude of the differences between algorithms compared. Therefore, an *effect size* test will be used to identify the strength of group differences as is recommended in quantitative research [63]. The calculation of the effect size based on the Mann-Whitney U test as a nonparametric effect size measure is expressed in Equation 5.1 from [80]. The z distribution (a.k.a, Standard Score (z-score)) is a signed number which indicates the difference between the value of observations and the population mean which can be reported by Mann-Whitney U test, and n is the total number of observations in data samples. The standard values of r are

that a large effect is 0.5, a medium effect is 0.3 and a small effect is 0.1, the absolute value is only reporting because the sign does not bear any extra information [80, 64, 37].

$$r = \frac{z}{\sqrt{n}} \tag{5.1}$$

Another technique to be considered is a box plot which is a powerful tool to convey statistical information about a single sample data [28]. The main reason for utilising box plots is that they depict groups of numerical variables through their quartiles as they are cut by horizontal lines through their medians so that the box plot is suitable for nonparametric tests since it makes use of the median instead of the mean and standard deviation to describe the population distribution [28]. Therefore, the box plots of data samples will be used to demonstrate which groups differ in the experiments. However, Bar Charts, which are a graphical representation of the mean, are used for the size-related performance. The rigorous combination of statistical procedures adopted in this research should support the validity of their results [37, 65]. Finally, all analyses were carried out using R version 3.2.2 [182]. The next section concludes this chapter by summarising its main contribution of it to the overall thesis's design.

5.6 Conclusion

To conclude, this chapter described the specification and guidelines for implementing the experimental framework in order to evaluate the performance, scalability and efficiency of the new approaches. The *holistic model* designed will be used to obtain experimental results by comparing different physical query plans in the query processor. Moreover, the platform set up was discussed in Section 5.3, and testing the framework was considered to validate the implementation of holistic twig matching methods. Then, an overview of the well-known XML datasets was given in Section 5.4. In addition, the appropriate statistical analyses were addressed to choose the most suitable tests in order to develop reliable experimental results over time-related performances. Overall, this chapter serves as a base for overcoming problems concerning reproducible experimental research in Computer Science by providing sufficient information to verify and reproduce the computational experiments carried out in this research [104, 202].

The following chapter discusses the new approach to improve the process of TPQs with Parent-Child edges relying on *Child Prime Labels* to process XML TPQs in a top-down processing manner. In addition, the chapter describes the process and implementation for assigning *Child Prime Labels* to internal XML tree nodes. The chapter is concluded with an experiment evaluation to compare the approach proposed with its competitors.

Chapter 6

Top-Down Approach based on Child Prime Labels

6.1 Introduction

In the recent years, many algorithms for processing twig pattern queries (TPQs) have been proposed in the literature [56, 40, 87, 5, 147]. A labelling scheme is fundamental to processing XML queries efficiently. They are used to determine structural relationships between elements corresponding to query nodes in TPQs. As was discussed in Chapter 4, increasing the meta-data of information contained with labels of XML elements may improve the filtering phase in holistic approaches.

This chapter starts by presenting a new indexing technique which exploits the property of *prime numbers* to identify Parent-Child (P-C) edges in TPQs during query processing. Two different approaches to index internal elements will be proposed. Then, it discuss the development of a novel holistic twig matching algorithm based on the new indexing approach for processing TPQs with Parent-Child (P-C) and Ancestor-Descendant (A-D) relationships efficiently. After that, a set of experiments to test the hypothesis introduced in Chapter 4 are described and used to evaluate the performance, scalability and efficiency of the new holistic algorithm.

The rest of this chapter is structured as follows. Section 6.2 covers some preliminaries including the notation and data structures in holistic algorithms and the limitations of TwigStack. The new indexing technique will be presented in Section 6.3. The new holistic algorithm will be introduced in Section 6.4. Section 6.5 presents the experimental evaluation and reports the performance comparison between the new algorithm and the previous comparable approaches. Finally, Section 6.6 concludes this chapter.

6.2 Preliminaries

6.2.1 Notation

Throughout this chapter, the term *element* will refer to a data element in an XML tree and *node* will refer a query node in twig pattern. As was discussed in the literature chapters, top-down holistic algorithms perform two phases. In the first phase, potential single paths are generated as intermediate results where each individual path corresponds to a non-predicate (i.e., root-to-leaf path) path expression of the query. The second phase consists of merge operations to stitch intermediate paths together and eliminate useless results. There are also some auxiliary operations on TPQ and its nodes to facilitate the twig matching process. Supported operations are as follows: *children(q)* returns all child nodes of q. *subtree(q)* returns all child nodes which are in the subtree rooted at q. *childrenAD(q)* returns all child nodes which have A-D relationship with q. *childrenPC(q)* returns all child nodes which have P-C relationship with q. *isRoot(q)* returns boolean values to see whether q is the root or not. *getRoot(TPQ)* returns the query root of the input TPQ. *parent(q)* returns the parent query node of q. *isLeaf(q)* returns boolean values to see whether q is a leaf node or not. In addition, previous top-down holistic approaches make use of a *Tag Streaming* scheme where each query node q is associated with a stream T_q (see Section 3.3.2.2) consisting of all elements with the same label as q in which the elements are sorted by the *start* values of their range-based labels in ascending order. It should be noted that elements may appear in different streams if there are nodes with the same tags in TPQ (i.e., similar node tests, see Section 5.2.2.1). Every stream T_q in TPQ is equipped with a cursor, denoted as C_q, which initially points to the first element in T_q at the beginning of a query processing. To ensure the linear processing in the filtering phase of holistic algorithms, only the first element is accessible and the rest elements are unseen by the algorithms, the filtering task has to be performed in a single forward scan of the streams. As shown in Figure 6.1, the stream of query node q has two parts *head* which is pointed by C_q while the remaining set of elements is referred to as *tail*. In order to accomplish the evaluation of the whole TPQ, the previous approaches augment the streams with virtual end element labelled with infinity values as (∞, ∞, ∞). The following operations are defined over every cursor of a stream in TPQ. *getStart(C_q)* returns the start attribute of the head element corresponding to query node q. *getEnd(C_q)* returns the end attribute of the head element corresponding to query node q. *getLevel(C_q)* returns the level attribute of the head element corresponding to query node q. *advance(C_q)* forward the cursor of q by one position to point to the next element. $eof(T_q)$ returns boolean values to judge whether or not C_q points to the end of stream of T_q.

Furthermore, stacks are fundamental data structures to twig matching algorithms [88]. In holistic algorithms, each query q is allocated to a stack named S_q, and each item of the stack consists of a pair: (the label of element,pointer points the matching item in the parent stack $S_{parent(q)}$). Moreover, each element in the stack is associated with two linked

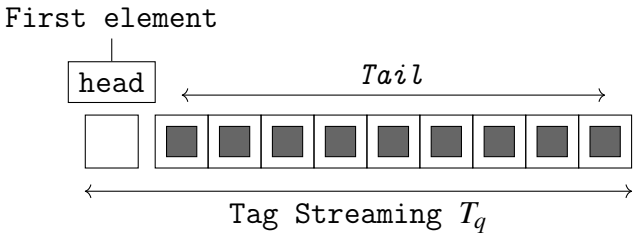


Figure 6.1: Tag Streaming Model of a query node q.

lists. The first list to represent all blocked descendant extensions (see Chapter 3) rooted at that element. The second list is to represent all blocked descendant extensions rooted at element which is a descendant that element [5, 40]. The common stack operations, such as *empty(), push(), pop() and top()*, are used. At any point during query processing, elements in stack S_q are nested from the bottom to top in a path containing the similar chain of elements as appear in the XML tree, and a chain of linked stacks is used to represent compactly intermediate results of individual root-leaf paths in TPQ.

The following section will discuss the limitations of TwigStack algorithm with respect to P-C edges in TPQs. It will also demonstrate important definitions to classify the state of the head elements of streams in XML query processing. Simple examples will be presented to show how the new indexing technique can be proposed.

6.2.2 Motivation and Limitations of TwigStack

In this section, a basic notion of optimal processing of TPQs in holistic approaches is discussed. Accessing head elements of streams is a fundamental property of holistic approaches, thus the original work of Bruno et al.[40] introduced a new definition which can control the size of intermediate results. That is an element e which corresponds to a query node q is pushed to the stack only if it has a *descendant extension* (see Section 4.2.1) which means that no element is stored in the stack unless it has useful descendant elements in the head elements of streams corresponding to its child query nodes and they recursively, in turn, have descendant extensions for sub-twigs rooted in them. Because of the restricted access mechanism, the holistic algorithms can not guarantee that the head elements would form matches to TPQs comprising of P-C edges. This dilemma (i.e., two head elements block each other with respect to a binary P-C relationship) causes the holistic algorithms to decide whether to output useless intermediate results or to miss some potential answers to TPQs. The researchers [54, 89] in holistic twig matching classified head elements pointed by cursors of streams to three types with respect to TPQ Q as follows:

1. Matching element: element e_n of a query node q_n is called a matching element if it has minimal extension to q_n as in Definition 6.1.

2. Useless element: element e_n of a query node q_n is called a useless element if e_n can not participate in a match to Q with the current or future elements.

3. Blocked element: otherwise element e_n of a query node q_n is a blocked element.

Definition 6.1 (Minimal Extension)**.** *Consider the head element e_n, pointed by C_{q_n}, which corresponds to query node $q_n \in TPQ\ Q$. e_n has a minimal extension if there is a strict subtree match to a subtree of Q rooted by q_n as in Definition 4.14, where every element of the match is the head element of its stream.*

It can be seen from the above definitions, TwigStack can guarantee linear processing for TPQs with only A-D edges, due to the fact that minimal extension can be always identified from the head elements forming matches to the query while useless elements can be discarded safely using the positional information of elements. It has been proven [40, 144, 61] blocked elements can not exist in TPQs with A-D relationships. The following examples illustrates the different head elements and minimal extension.

Example 6.2. *Consider the binary structural relationship A-D between two head elements in the streams as e_a and e_d of the query nodes q_a and q_d, respectively, representing simple path query a//d. There are four cases which describe the status of the current head elements as illustrated in Table 6.1. The possible cases of the head elements when the structural relationship between them is P-C is presented in Table 6.2 for the head elements e_p and e_c corresponding to the query nodes q_p and q_c, respectively for the path expression p/c.*

Example 6.3. *Consider the XML tree T_1 in Figure 6.2 and the TPQ Q_1 a[//x]/y/f, at the beginning of query processing the head elements are $C_{q_a} \rightarrow a_1$, $C_{q_x} \rightarrow x_1$, $C_{q_y} \rightarrow y_1$ and $C_{q_f} \rightarrow f_1$. The element y_1 has a minimal extension to the subtree rooted at the query node q_y as (y_1, f_1) which is the match to the sub-twig of q_y. The head element a_1 does not have a minimal extension because it is a blocked element since a_1 and y_1 block each other as described in Table 6.2 so that a blocked element has to be stored in order to compute a match (if any) to Q_1.*

The existence of blocked elements influences the optimality of holistic twig matching algorithms, an optimal holistic twig matching algorithm should not have blocked elements, in other word, the head elements are either matching or useless. Therefore, streams can be forwarded without storing irrelevant elements. The next section presents intuitive example to illustrate different classes of head elements and suboptimal evaluation for TPQs using the information contained within the existing labels.

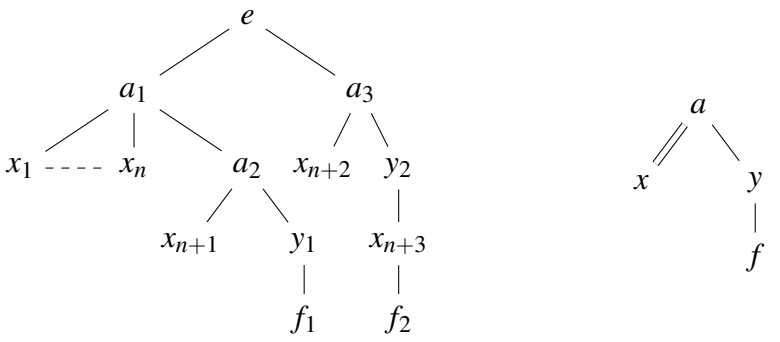
6.2.2.1 Straightforward Example

This section illustrates the main problem of TwigStack which assumes edges in TPQs are A-D. It shows also how the number of blocked elements in streaming lists, when P-C

Table 6.1: Possible cases for binary structural A-D relationship shown in a//d.

	Case 1	Case 2	Case 3	Case 4
Property	getEnd(e_a) < getStart(e_d)	getStart(e_a) < getStart(e_d) and getEnd(e_a) > getEnd(e_d)	getStart(e_a) > getEnd(e_d)	getStart(e_a) > getStart(e_d) and getEnd(e_a) < getEnd(e_d)
e_a	useless element	matching element	blocked element	blocked element
e_d	blocked element	matching element	useless element	useless element

relationships involved in the query, influences the optimal evaluation for TPQs in holistic approaches. These limitations will be demonstrated in the following example.



(a) an XML tree T_1.

(b) a twig query Q_1.

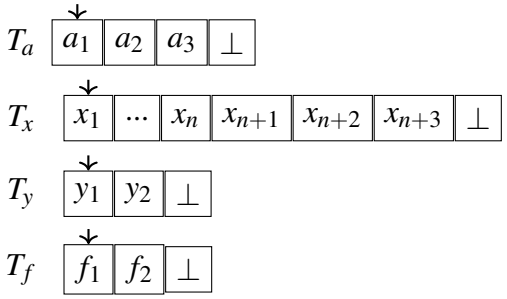
Figure 6.2: Illustration of the suboptimal processing of TwigStack.

Example 6.4. *Consider the XML tree T_1 and the TPQ Q_1 a[//x]/y/f of Figure 6.2. The first cycle of TwigStack identifies that the element a_1 is blocked (see case 3 in Table 6.2)and it is pushed into the corresponding stack, and the table in Figure 6.3 presents the situation at each iteration of TwigStack. After n iterations, the n number of x has been processed, the algorithm generated n single paths corresponding to the simple path expression a//x. After n+1 cycle, the element a_2 is found to have a minimal extension and can be returned to compute a match to Q_1 since it corresponds to the query root. This match is confirmed by the main algorithm at iteration n+4 after the element f_1 is pushed into its query node stack. Then, TwigStack proceeds to the next head element to see whether the element a_3 has a descendant extension or not. From the head elements, it can be seen the element a_3 has the descendant extension but two head elements are found to be blocked and the algorithm can no longer guarantee their contributions to the result of Q_1, namely a_3 and y_2. Finally, at the iteration n+9, the algorithm can ensure that the element f_2 does not participate in the final result as it does not have P-C relationship with y_2 which is the top element in its parent stack as $S_{parent(f)}$. By this step, TwigStack has produced two more useless paths as (a_3, x_{n+2}) (a_3, x_{n+3}). Note that there is only one match to the TPQ Q_1 in this example but TwigStack generated n+3 useless intermediate paths.*

As was discussed in Chapters 3 and 4, the number of intermediate results has a significant impact on the performance holistic twig matching algorithms and the above example has shown that TwigStack can not filter out irrelevant elements for TPQs with P-C edges. In the worst case, the number of intermediate root-to-leaf paths can be $O(F \times L)$

Table 6.2: Possible cases for binary structural P-C relationship shown in p/c.

	Case 1	Case 2	Case 3	Case 4	Case 5
Property	Case 1 in A-D	Case 2 in A-D and getLevel(e_p) - getLevel(e_c) = 1	Case 2 in A-D and getLevel(e_p) - getLevel(e_c) > 1	Case 3 in A-D	Case 4 in A-D
e_p	useless element	matching element	blocked element	blocked element	blocked element
e_c	blocked element	matching element	blocked element	useless element	useless element



(a) Streams and arrows indicating the head elements.

iteration	returned node	head elements	key element	Operations
1	a	a_1, x_1, y_1, f_1	a_1 is blocked	push a_1
...	...	...	...	...
n	x	a_2, x_n, y_1, f_1	x_n *is useless*	push x_1
n+1	a	a_2, x_{n+1}, y_1, f_1	a_2 has a minimal extension	push a_2
n+2	x	a_3, x_{n+1}, y_1, f_1	x_{n+1} *is useless*	push x_{n+1}
n+3	y	a_3, x_{n+2}, y_1, f_1	y_1 *is useless*	push y_1
n+4	f	a_3, x_{n+2}, y_2, f_1	f_1 *is useless*	push f_1
n+5	a	a_3, x_{n+2}, y_2, f_2	y_2 is blocked	pop a_2 *and* a_1 , push a_3
n+7	y	$\perp$, x_{n+3}, y_2, f_2	y_2 is blocked, x_{n+3} is useless	pop y_1 , push y_2
n+8	x	$\perp$, x_{n+3}, $\perp$, f_2	x_{n+3} is useless	pop x_{n+2} , push x_{n+3}
n+9	f	$\perp$, $\perp$, $\perp$, f_2	f_2 is useless	discard f_2

(b) TwigStack processing of Q_1.

Figure 6.3: Illustration of TwigStack operations.

where F is the sum of the lengths of the input lists for leaf query nodes and L is the maximum depth in the XML tree [61, 144, 116]. It should be noted that an element might be useless with respect to the current head elements but it may participate in a match to the query if its ancestor is stored in the corresponding stack. TwigStack applies a *strict prefix-filtering* (see Definition 4.16) to handle useless elements returned by *getNext()*. Furthermore, Example 6.4 illustrates the main drawback of TwigStack which may be a result of the limited access only to the head elements in which there is no such a way to find elements which satisfy Parent-Child relationships with the rest of the input streaming lists.

Having illustrated the limitations in the evaluation of the head elements, the next sections will discuss the proposal for a new indexing technique to overcome blocked elements issues with P-C edges. The development of a new holistic twig matching algorithm based on the indexing mechanism proposed will be discussed as well.

6.3 Child Prime Labels

This section presents a new indexing technique which can be applied to the existing labelling schemes to minimise the number of blocked cases in the streams during the processing of TPQs with Parent-Child axes. The key idea of the work presented in this section is to find an appropriate technique which can be used in addition to the triplet of

range-based labelling scheme to resolve Case 3 in Table 6.2. The name of the new approach, *Child Prime Labels (for short CPL)*, is driven from the exploitation of child relationships in XML trees (see Definition 4.6) and the property of prime numbers. Therefore, the immediate child elements of inner elements can be derived from their labels, then the process to handle Parent-Child relationship among head elements in the streams can be resolved by computation.

The idea is to identify all the distinct tags in the XML tree and assign them with unique prime numbers. Then, the intuition of the CPL is to use the modulo function to test whether an element has a particular element name among its children. The leaf elements will be annotated with 1 as their CPLs, while the inner elements (i.e., parent elements) are assigned CPLs by multiplying the prime numbers of its distinct names of child elements. For illustration, consider an element e, with all distinct names of children, $C = \{c_1, c_2, \ldots, c_m\}$ and a list of prime numbers $P = \{p_1, p_2, \ldots, p_m\}$. The bijective mapping function $f : C \rightarrow P$ for all element $p \in P$, there is a unique element $c \in C$ such that $f(c) = p$. Then, the CPL for element e can be computed as follows:

$$CPL(e) = \begin{cases} \prod_{i=1}^{m} f(c_i), & \text{if } m \geq 1 \\ 1, & \text{otherwise} \end{cases} \tag{6.1}$$

This thesis aims to extend the original range-based labelling scheme to incorporate the CPL information see Definition 6.7. Each range-based label with CPL is presented as quadruple *=(start,end,level,CPL)*. The first three attributes remain the same as in the original labelling scheme see Chapter 5. According to Proposition 6.5, all distinct names of immediate child elements for a particular element in the XML tree can be obtained from having the corresponding prime numbers associated with tag names of its children. For efficient generation the CPL labels, they can be generated during the preorder traversal of the XML tree. Algorithm 3 presents the process for CPL generator. The algorithm can be seen as an extension of Algorithm 1 in Chapter 5. It first initialises a stack to record CPL for the current processing element. A *hash* table is used to create a mapping from an element tag to a prime number as in Equation 6.1. During depth-first scanning, the current element is assigned the next available prime number if its tag has not been examined (Lines 15-18). The smallest prime number is 2 because 1 is reserved to identify leaf elements in the XML tree as explained in the equation above. After that, Lines 20-26 check the CPL parameter of its parent element to see whether it is divisible by the assigned prime number or not. If it is not a factor of the parent CPL, the parent element's CPL is multiplied by the new prime number after checking overflow condition. In case of overflow, the CPL has a value of 0 to avoid lose potential elements, in other words there is no change to the original labelling scheme. Otherwise, Line 26 assigns 1 as CPL for the current element since it appears before its child elements (if exist). Line 28 handles special case for the root element. When reading the closing tag of the current element, *endElement()* function generates the complete label for the element under investigation as 4-tuple in which the

CPL parameters are 1 for leaf elements, while parent elements are assigned their CPL computed by popping the global stack, *childStack* in Lines 30-35.

Proposition 6.5 (Uniqueness [166])**.** *There is only one unique set of prime factors for any number.*

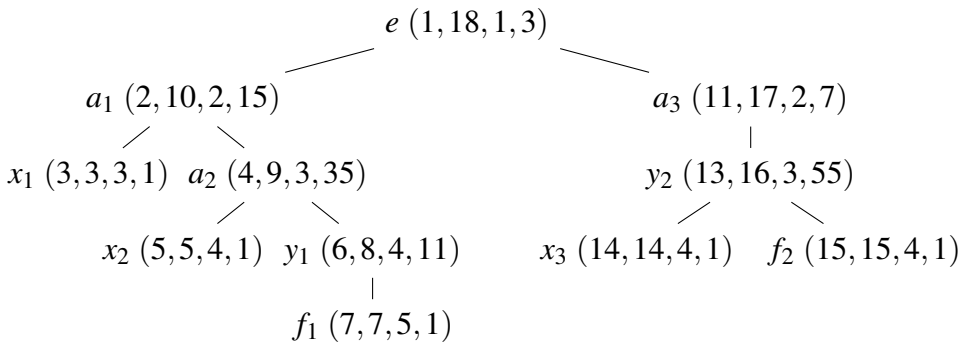
Proposition 6.6 (Zero Factors [166])**.** *Zero is a multiple of any number.*

According to Proposition 6.6, overflowing problems in CPLs can be handled without modifying the query results. In case of overflow, the information obtained from CPLs can be discarded and holistic twig matching algorithms based on CPL would continue processing current head elements using the original range-based labelling scheme. However, five different datasets (see Section 5.4) were used in the experiments of this thesis and there was no sign of overflow.

Definition 6.7 (Child Prime Label)**.** *A child prime label is assigned to each element in an XML document as an extra parameter into the range-based label. A child prime label indicates the multiplication of distinct prime numbers for every internal elements within the document, and leaf elements are assigned 1.*

Example 6.8. *Consider the XML tree in Figure 6.4. For instance the CPL of the root element e is computed as follows. When reading the opening tag of e, it is given 1 as its initial CPL and it is pushed into a global stack and stored in tag indexing. After scanning the element*a_1*, the tag a is assigned with 3 as its unique prime number. The CPL of e is checked to see whether it is divisible by the new prime number or not. Since the old value 1 is not divisible by 3, the CPL of e is updated to* 1×3*. Once the element* a_3 *is received, its tag is checked to see whether it has been assigned a prime number or not. The tag a has already been assigned a prime number by looking up the tag indexing dictionary. The CPL of its parent element e is examined to see whether the prime number of* a_3*'s tag divides e's CPL or not. Because 3, as the prime number of tag a, is already a divisor of the CPL of e which is 3, the algorithm proceeds without updating the CPL of e. In like manner, the element* y_2 *has a CPL with value of 55 as* $CPL(y_2) = f(x) \times f(f) = 5 \times 11 = 55$.

It can be seen from the above example, the process of assigning prime numbers to distinct tags in the XML tree is simple and straightforward. It can be performed in a single scan of the original document. However, assigning small prime numbers to the top elements in the tree may result in relatively large numbers assigned to CPLs. This may also increase the number of CPLs which overflow, thus it has a detrimental impact on the advantages of CPLs. An improved approach to assign unique prime number to the distinct tag names in the XML tree is proposed to reduce the number of overflow cases. In the improved approach, the labelling scheme is done in two phases. First, it traverses the XML tree once to determine for each distinct tag name the number of distinct sibling tags, which is used later to sort tag names according to their number of distinct sibling tags in descending order. The sorted tag names are assigned prime numbers from the smallest



(a) an XML tree T_1.

Tagname	Key
e	2
a	3
x	5
y	7
f	11

(b) tag index.

Figure 6.4: An XML tree labelled with range-based augmented with CPL and the corresponding tag indexing.

to the largest. In the second phase, the result of the first phase is used to initialise the *hash* table in Line 4 of Algorithm 3, and Algorithm 3 scans the document in depth-first traversal to label the XML elements. The Algorithm 4 illustrates the process for counting the number of distinct sibling tag names and pairing tag names with prime number. The following example shows that the improved approach can avoid assigning small prime numbers to tag names corresponding to elements at the top of the XML tree.

Example 6.9. *Consider the XML tree in Figure 6.4. The tag names are assigned prime numbers as they have been seen by the SAX parser in depth-first traversal. The order for assigning prime numbers can be presented as follows:* $e \rightarrow 2$, $a \rightarrow 3$, $x \rightarrow 5$, $y \rightarrow 7$, $f \rightarrow 11$. *Using the improved approach, the order of assigning prime numbers is different since it considers the number of sibling tags. The same XML tree is labelled by the improved approach in Figure 6.5. For instance, the tag name x is given the smallest prime number because it has the maximum number of distinct sibling tags.*

The next section describes the advantages of the CPL. Furthermore, it discusses the characteristics of the CPL in terms of handling Parent-Child relationships in the streaming model.

6.3.1 Properties of Child Prime Label

This section summarises the properties of Child Prime Labels indexing. The CPL of a particular element can be used to derive a set of the tag names of its immediate child elements as in Property 6.10. For sake of clarity, two more operations are defined on

Algorithm 3: CPL-Region Encoding Algorithm

Input: an XML file xml

Result: XML elements labelled with the range-based labelling scheme and augmented with CPL where internal elements are assigned the product of their children's prime numbers while leaf elements assigned 1

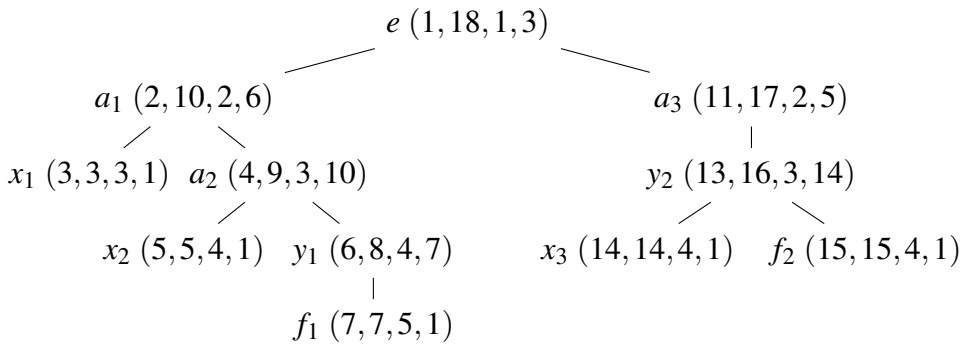
```
1  // initialization
2  // see Algorithm 1
3  IntegerStack childStack = ∅ // stack holds the number of children for each element.
4  HashTable tagIndexing = ∅
5  // hash table to generate string to Integer mapping where tags are keys and prime
     number are values, and produce
6  tag indexing to be used later by query processors, see Section 5.2.1.4
7  order = 0 // document order level = 1 // level of element
8  while ¬ read(xml) do
9      // if opening tag calls startElement
10     // if closing tag calls endelement
11 Procedure startElement(tag,attributes):
12     order = order + 1 // generate sequential integer number.
13     push(orderStack,order)
14     level = level + 1
15     if ¬contain(tagIndexing,tag) then
16         currentPrime = getPrime() // returns the next smallest prime number to be
             used for the current element starting from 2 as the smallest prime number
17         put(tagIndexing,tag,currentPrime)
18         // insert the prime to the disctionry
19     if ¬ isEmpty(childStack) then
20         CPL = pop(childStack)
21         // check if the CPL of the parent element is divisible by the prime of its new
             child tag
22         if x mod get(tagIndexing,tag) ≠ 0 then
23             CPL = CPL × get(tagIndexing,tag)
24             // here apply safeMultiple to avoid overflow, in case of overflow CPL = 0
                 to ensure the element is not mistakenly discarded by twig matching
                 algorithms
25             push(childStack,CPL) // update the parent CPL
26         push(childStack,1) // creat the CPL for the current element
27     else
28         push(childStack,1) // this is for the root node because it does not have a
             parent.
29 Function endElement(tag):
30     level = level - 1 // return to the previous level
31     if top(childStack) ≠ 1 then
32         order = order + 1
           return :(pop(orderStack),order,level,pop(childStack))
33
34     else
           return :(pop(orderStack),order,level,1)
35
```

Algorithm 4: Generate Tag Indexing Algorithm

Input: an XML file xml
Result: XML tags with the largest number of distinct sibling tags are assigned the smallest prime numbers

```
1  // initialization
2  IntegerStack childStack = ∅ // stack holds the number of children for each element.
3  HashTable siblingCount = ∅ // the key is string and the value is a list of string
4  // hash table to compute number of distinct tags for each tag in the XML tree
5  // each entry of the table returns a list of string to compute the number of distinct tags

6  elementStack eStack = ∅
7  // stack to hold objects of elements which contains a list, called cList to store distinct
     child tags)
8  while ¬ read(xml) do
9      // if opening tag calls startElement
10     // if closing tag calls endelement
11 tagList = sorted keys of siblingCount based on the sizes of their sibling lists
12 HashTable tagIndexing = ∅
13 foreach tag in tagList do
14     currentPrime = getPrime() // tag is paired with getPrime() returns the next
         smallest prime number to be used for the current element according to the order
         based on the number of distinct tags to mitigate overflow issues
15     put(tagIndexing,tag,currentPrime)
   Return : tagIndexing // Tag Indexing to be used during the labelling process in
            Algorithm 3 at Line 5
16 Procedure startElement(tag,attributes):
17     if ¬ isEmpty(childStack) then
18         x = pop(childStack)
19         x = x+1
20         push(childStack,x) // for the parent element
21         push(childStack,1) // for the current element
22         if ¬top(eStack).cList.contains(tag) then
23             top(eStack).cList.add(tag) // add the new tag to the list of child tags
24     else
25         push(childStack,1) // this is for the root node because it does not have a
             parent.
26     e = new Object
27     push(eStack,e)
28
29 Procedure endElement(tag):
30     // if the current element has a child element, the distinct tags are calculated
31     if pop(childStack) ≠ 1 then
32         foreach child in top(eStack).cList do
33             List sList = get(siblingCount,child) // sibling list
34             foreach sibling in top(eStack).cList do
35                 if ¬ sList.contains(sibling) then
36                     sList.add(sibling)
37             replace(siblingCount,child,sList)
38             // update the list of sibling tags to compute the total number at the end
```



(a) an XML tree T_1.

Tag	Key
e	11
a	3
x	2
y	5
f	7

(b) tag index.

Tagname	# of sibling tags
e	0
a	1
x	3
y	1
f	1

(c) First phase in the improved approach.

Figure 6.5: The improved approach to label the XML tree in Figure 6.4.

query nodes of TPQs and over cursors during the query processing as in Section 6.2.1. *tagPrime(q)* returns the unique prime number associated with q from *tag indexing* (see Definition 6.12). *getCPL(C_q)* returns the CPL attribute of the head element corresponding to query node q.

Property 6.10 (CPL Relationship)**.** *In any XML labelling scheme that is augmented with Child Prime Labels, for any elements x,y and z in an XML document, x has at least one or more child elements of label $\mu(y)$ and $\mu(z)$ if and only if CPL_x mod $key_{\mu(y)} \times key_{\mu(z)} = 0$, where $key_{\mu(y)}$ and $key_{\mu(z)}$ are defined prime numbers.*

In addition to the three classes of head elements described in Section 6.2.2, this thesis introduces a new class of head elements with respect to TPQ Q as follows:

4. Possible matching element: element e_n of a query node q_n is called a possible matching element if it has possible extension to q_n as in Definition 6.11.

Definition 6.11 (Possible Extension)**.** *Consider the head element e_n, pointed by C_{q_n}, which corresponds to query node $q_n \in TPQ\ Q$. e_n has a possible extension if there is a semi-strict subtree match to a subtree of Q rooted by q_n as in Definitions 4.14 and 4.17, and the information derived from CPL of e_n satisfies the CPL relationship for each child query node.*

Definition 6.12 (Tag Indexing)**.** *The tag indexing is a lookup table to find unique prime numbers associated with distinct tags within a given XML document during query processing.*

The original range-based labelling scheme does not have Property 6.10 so that there are only five possible cases for Parent-Child relationships between two head elements in the streams. Case 3 in Table 6.2 is associated with suboptimal evaluation of TPQs containing a combination of A-D and P-C axes [144, 54]. Utilising the properties of CPLs, the present study further classifies Case 3 in Table 6.2 into two sub-cases which are illustrated in Table 6.3. Case 3-1 changes the state of the parent query node to be useless rather than blocked if the parent element does not satisfy the CPL relationship with the child element (see Property 6.10). In Case 3-2, the parent element is found to satisfy the CPL relationship and it can be expected to have at least one child element in the tail of its child input stream.

In the following section, the Child Prime Labels will be used to design a novel top-down holistic twig matching algorithm which utilises Property 6.10 to reduce the intermediate results by decreasing the number of blocked head elements in the streams during the query processing.

6.4 Holistic Twig Matching Algorithm with Child Prime Label

Having established the concept of CPL indexing, the present research proposes a new top-down holistic twig matching algorithm using the CPL. This approach differs from the previous algorithms mainly in the labelling used.

6.4.1 Top-Down Twig Matching Algorithm: TwigStackPrime

This section present a new top-down holistic twig matching algorithm, called TwigStackPrime. The new approach can be seen as an alternative to TwigStack algorithm. The original TwigStack remains the same with the only difference being in the advanced pre-order function *getNext*. The use of stacks in TwigStackPrime is similar to that in TwigStack (see Section 6.2.1).

The structure of the main algorithm, *TwigStackPrime* presented in Algorithm 6 is not much different from the original holistic twig join algorithm *TwigStack* [40] which uses two phases to compute answers to a twig query. In the first phase, solutions to root-to-leaf paths in a TPQ are found and stored in output arrays (Lines 1-11). It repeatedly calls the *getNext* algorithm (see Algorithm 5) with the query root as the parameter to return the next

Table 6.3: Further classification of head elements for p/c in Table 6.2.

	Case 3-1	Case 3-2
Property	Case 2 in A-D and getLevel(e_p) - getLevel(e_c) > 1 and getCPL(e_p) mod tagPrime(q_c) $\neq 0$	Case 2 in A-D and getLevel(e_p) - getLevel(e_c) > 1 and getCPL(e_p) mod tagPrime(q_c) == 0
e_p	useless element	possible matching element
e_c	blocked element	blocked element

query node for processing. In the second phase (Line 12), solutions in the output arrays are merge-joined based on their common branching query nodes and query matches are returned as the query result. The number of output arrays is equal to the number of leaf query nodes (i.e., the number of individual root-to-leaf paths in a TPQ).

getNext is an essential function which is called by the main algorithm to decide the next query node to be processed. It is fundamental to guarantee that the current head element associated with the query node returned is part of the final output since all the basic structural relationships are thoroughly checked by *getNext* or its supporting subroutine *getElement*. *getNext(q)* returns an element e_q of a query node $q \in TPQ$ with three properties:

i e_q has a descendant element e_{q_i} in each of the streams corresponding to its child elements where e_{q_i} is the head element of a query node q_i = *children(q)* (this property is checked in Lines 9-11).

ii each of its child elements satisfies recursively the first property (this property is checked in Lines 4-5).

iii if q has Parent-Child edge(s) with its child query nodes, then e_q has a child e_{q_i} in T_{q_i} for each query node q_{q_i} = childrenPC(q) (this property is checked in Lines 21-23 of *getElement* function).

Moving on now to go through *getNext* algorithm. Firstly, if q is a leaf query node, it trivially satisfies the three properties, thus it is returned in Line 1. After that, *getNext* invokes recursively *getNext* for each query node n_i = *children(q)*. If any query node g_i is not equal to n_i, g_i is immediately returned because n_i can not satisfy the aforementioned three properties. This is how the algorithm handles Cases 3 and 4 for A-D relationship between elements in the streams as presented in Table 6.1. The cursor pointed to g_i is advanced to point to the next element in the stream. Otherwise, every child of q satisfies the three properties. Lines 4 and 5 get the *max* and *min* the head elements corresponding to child query nodes of q. Line 7 discards elements of q which do not contribute to the final result.This line handles Case 1 for A-D (see Table 6.1). In Line 8, the first property is checked. If the current head element of q fails to satisfy the first property, the child query node with the smallest start value is returned in Line 9 (handling Cases 3 and 4 for A-D, see Table 6.1). Otherwise, the query node q is returned that means it corresponding to Case 2 for A-D or Cases 2 and 3-2 for P-C. In another way, the current head element of q is either matching or possible matching.

In the function *getElement(q)*, if q does not have P-C edges, the current head element of q is returned. Otherwise, Line 21 checks CPL relationship for all child query nodes with Parent-Child relationships. If the current element does not satisfy the CPL relationship (the third property), the function skips all elements which do not satisfy the CPL relationship (Case 3-1 for P-C in Table 6.3). Otherwise, the head element satisfying the CPL relationship

is returned in Line 21 if the stream is unfinished. In case the stream reaches the end, Line 19 returns virtual end element labelled with infinity values to complete the query processing.

Algorithm 5: getNext(q)

Input: q is a query node
Result: a query node in TPQ which may or may not be q

```
1  if isLeaf(q) then
       return : q
2  foreach node n_i in children(q) do
3      g_i = getNext(n_i) if g_i ≠ n_i then
           return : g_i
4  n_max = a query node with the maximum start value ∈ children(q)
5  n_min = a query node with the minimum start value ∈ children(q)
6  while getEnd(getElement(q)) < getStart(getElement(n_max)) do
7      advance(q)
8  if getStart(getElement(q)) < getStart(getElement(n_min)) then
       return : q
9  else
       return : n_min
10 Function getQCPL(Query node q):
11     // the prime number assigned to the query node which is the product of its child
         query node prime numbers, it is called only once for each branching query node
         with P-C edge(s)
12     qCPL = 1
13     foreach node n_i in childrenPC(q) do
14         qCPL = qCPL × tagPrime(n_i)
       return : qCPL
15 Function getElement(Query node q):
16     if childrenPC(q) > 0 then
17         while ¬ eof(C_q) ∧ getCPL(C_q) % getQCPL(q) ≠ 0 do
18             advance(q)
19     if eof(C_q) then
           return : ∞, ∞, ∞, 1 // out of range label
20
21     else
           return : C_q // the current head element in the stream of q
22
```

Turning now to the main algorithm of TwigStackPrime, it repeatedly calls *getNext* to get the next query node to be processed as follows. Firstly, Line 2 calls *getNext* with the root query node as the input parameter to identify the query node q_{act} to be processed. After that, Line 4 performs a strict prefix matching (see Definition 4.13) by removing elements which are not ancestors of the head element of q_{act} from the stack of $parent(q_{act})$. If the head element of q_{act} has the ancestor extension and q_{act} is not a leaf query node, it is pushed into its corresponding stack and assigned a pointer to its ancestor/parent element in the parent stack at Line 4. Otherwise, all individual root-to-leaf paths involving the head element are generated and stored in their output array. Note that some path solutions may be blocked using blocking sorting technique introduced in [5, 40] (see Chapter 3) in order

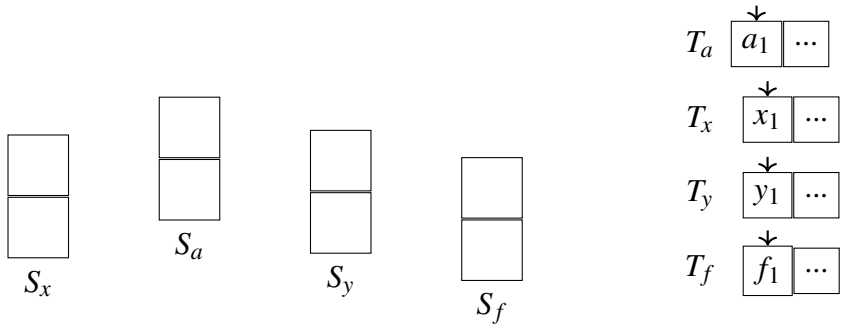
to sort simple paths in a sorted order of their common prefixes. If the element of q_{act} fails to satisfy the strict prefix the matching, its cursor is shifted to point to the next element in the stream and the algorithm proceeds to the next cycle.

Compared to TwigStack (i.e., any holistic algorithm which satisfies only the first and second property in *getNext* and does not apply CPL relationships), the effect of TwigStackPrime can be illustrated in the following examples.

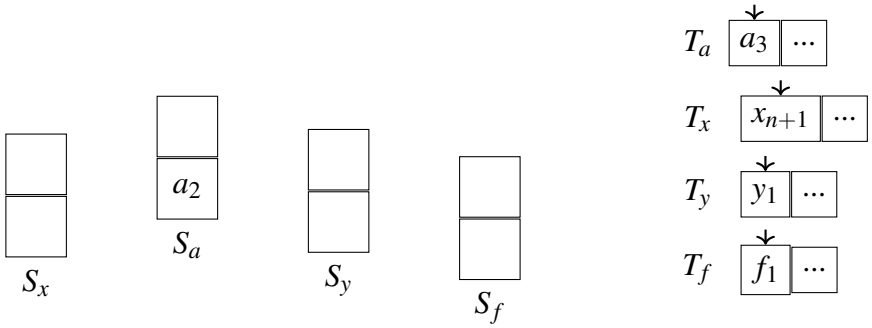
Example 6.13. *Consider the XML tree T_1 of Figure 6.2 and Q_1 = a[//x]/y/f. Recall that TwigStack generated n+3 useless paths. Assume the tree is labelled with range-based labelling and CPL indexing as in Figure 6.5. Initially, the cursors point at the head elements as depicted in Figure 6.6. getNext(a) is called since a is the root query node. The first call of getNext(a) in TwigStack returns a_1 because it satisfies the descendant extension condition, but TwigStackPrime skips a_1 since it does not satisfy the CPL relationship that is CPL of a_1 is not divisible by the prime number associated with the tag name y. The algorithm has to skip n elements with tag x since they are useless to the head element a_2. After this, TwigStackPrime can ensure that a_2 has a minimal match and thus is pushed into the stack of a. Next, a_3 is the head element of a, see Figure 6.6b, so that elements corresponding to query nodes x and y are pushed into their stacks to form partial paths with a_2. After n+3 iteration, a_3 is discarded from the stream because it does not satisfy the first and second property since y_2 is removed from its stream. For instance, $CPL(y_2) \rightarrow 3$ mod $tagPrime(f) \rightarrow 7$ is not equal to zero. The algorithm terminates after performing n+5 recursive calls of getNext(a).*

Example 6.14. *Consider the XML tree of Figure 6.7a and the TPQ Q_1 = a[//x]/y/f in Figure 6.2. The head elements in their streams are $C_a \rightarrow a_1$, $C_x \rightarrow x_1$, $C_y \rightarrow y_1$ and $C_f \rightarrow f_1$. The first call of getNext() inside the main algorithm will return $a \rightarrow a_1$ because it has A-D relationship with all head elements and satisfies the CPL relationship with x and y, and its descendant $y \rightarrow y_1$ also satisfies the child and descendant extension with respect to f, Figure 6.7 shows the stack encoding during the query processing of Q_1 and the output arrays corresponding to the individual root-to-leaf paths. However, TwigStackPrime generates the useless paths (a_1, x_1) and (a_1, x_2) because the element a_1 has a possible extension as in definition 6.11 and after scanning its child element y_2, it is found that it does not satisfy the three properties. There are only two merge-able paths which consist the final match. These paths are $(a_2, x_2), (a_2, y_1, f_1)$.*

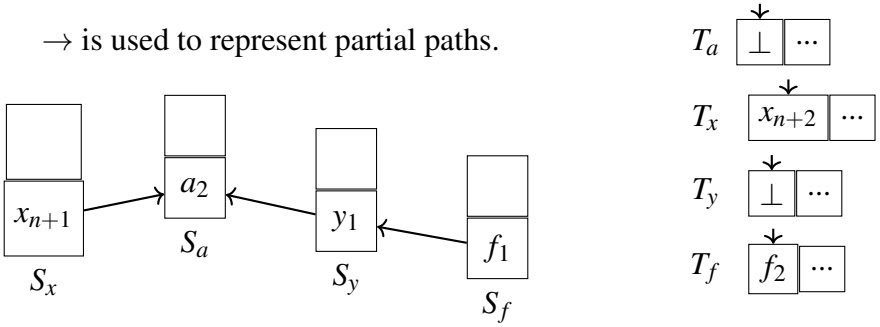
It can be seen from the above examples, the *getNext* proposed in this thesis can use the CPL relationship to eliminate irrelevant elements from the parent streams as was illustrated in Example 6.13. Due to the restricted access mechanism, TwigStackPrime may produce useless paths as it considers only the CPL relationship between two streams. Example 6.14 illustrates some cases when TwigStackPrime can not guarantee that all path solutions will contribute to the final result. However, TwigStackPrime is still superior to TwigStack in



(a) Stack encoding of TwigStackPrime at the first cycle.

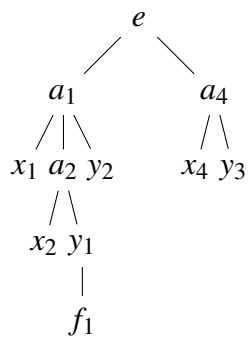


(b) Stack encoding at the n cycle.



(c) Stack encoding at the n+3 cycle.

Figure 6.6: Illustration to TwigStackPrime processing of Q_1 on T_1 in Figure 6.2.

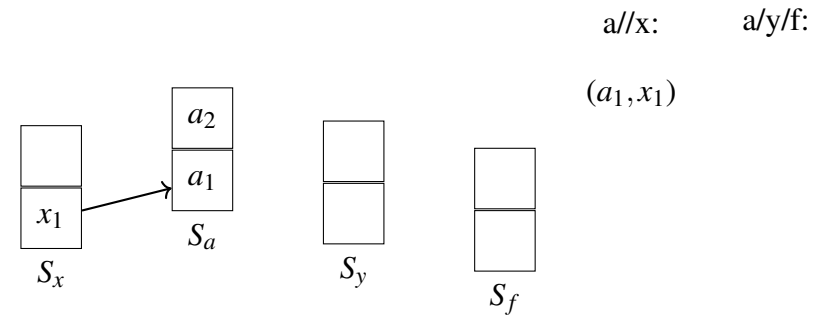


(a) an XML tree T_2.

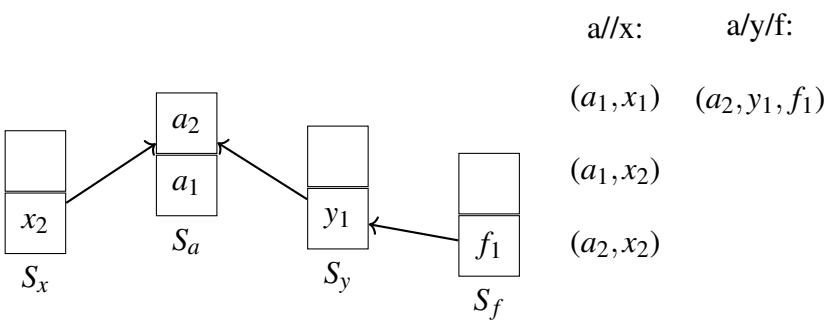
Figure 6.7: An example to illustrate a case when useless paths may be produced.

Algorithm 6: TwigStackPrime

```
   Input: TPQ Q
1  while ¬end(getRoot(Q)) do
2  |  q_act = getNext(getRoot(Q)) // see Algorithm 5
3  |  if ¬ isRoot(q_act) then
4  |  |  cleanStack(q_act, parent(q_act))
5  |  if isRoot(q_act) ∨ ¬ empty(S_parent(q_act)) then
6  |  |  cleanStack(q_act,q_act)
7  |  |  moveToStack(q_act)
8  |  |  if isLeaf(q_act) then
9  |  |  |  outPathSolution(q_act) // Blocked solutions as introduced in [40]
10 |  advance(q_act)
11 MergeAllPathSolutions() // Phase 2 as introduced in [40]
12 Procedure cleanStack(Query node q_act,Query node q):
13 |  // pop any element in S_q which is not the ancestor of getElement(q_act)
14 |  while ¬empty(S_q) ∧ getEnd(top(S_q)) < getStart(getElement(q_act)) do
15 |  |  pop(S_q)
16 Procedure moveToStack(Query node q):
17 |  // p is a pointer to the top parent stack if q is the root p is null
18 |  // p = top(S_parent(q))
19 |  push(C_q, p) to S_q
20 Function end(Query node q):
   |  return : ∀n_i ∈ subtree(q) : isLeaf(n_i) ∧ eof(C_ni)
```



(a) Stack encoding before x_2 is pushed into S_x.



(b) Stack encoding when f_1 is pushed into S_f.

Figure 6.8: Illustration to TwigStackPrime processing of Q_1 on T_2 in Figure 6.7a.

terms of the number of intermediate single paths as it will be shown in the experimental section 6.5.

Before proceeding to show the correctness of TwigStackPrime algorithm, it is important to compare it with TwigStackList [144] which uses a simple buffering technique to cache some elements from the parent streams in order to resolve blocked head elements as described in Case 3 of Table 6.2. The difference between the algorithms is presented in the following example.

Example 6.15. *Consider the XML tree T_3 of Figure 6.9a, and a TPQ Q_2 = a[//x]//y/f in Figure 6.9b. The first call of getNext(a) in TwigStackList returns a_1 but before that the algorithm buffers all elements with tag name y which are ancestors of the head elements of query node f to buffering list. The buffering list contains elements from y_1 to y_n and the cursor pointed to y_n because the head element of query node f is its immediate child element, thus, $C_y \rightarrow y_n$. TwigStackList returns elements y_n and f_1 before y_1 in order to move the cursor the next element of query node f. Therefore, TwigStackList produces path solutions which are not ordered in a sorted order of their common prefixes as (a_1, y_n, f_1) precedes (a_1, y_1, f_2) which is incorrect according to the definition of twig matching result (see Definition 4.11). Unlike TwigStackList, TwigStackPrime doe not violate the document order. The first call of getNext(a) returns a_1 and the second iteration return y_1 because it has a child and descendant extension. Thus, the path solutions generated are (a_1, y_1, f_2) followed by (a_1, y_n, f_1) which can be merged with the path (a_1, x_1) to compute matches to Q_2. Finally, the individual root-to-leaf paths are merged together based on their common branching query node a to form the matches to Q_2. Therefore, the query result consists of two matches (a_1, x_1, y_1, f_2) and (a_1, x_1, y_n, f_1).*

Example 6.15 demonstrates the effect of TwigStackPrime in filtering useless elements without consuming extra storage and manipulating node processing order, see Section 4.2.3. The next section presents definitions and theorems to prove the correctness of TwigStackPrime.

6.4.2 Analysis of TwigStackPrime

This section shows the correctness of TwigStackPrime and analyses its complexity. The correctness of *TwigStackPrime* algorithm can be shown analogously to *TwigStack* due to the fact that they both use the same stack mechanism. In other words, the correctness of Algorithm 6 follows from the correctness of *TwigStack* [40]. Therefore, if the algorithm is optimal, the space, time and I/O complexity is the same as shown in TwigStack [40]. Recall that, TwigStack was reported to has a linear worst-case I/O complexity with respect to the result size when TPQs contain only A-D edges. Since the use of CPL properties in *getNext* improves the filtering strategy, this section aims mainly to prove its correctness.

Definition 6.16 (Head element)**.** *For each query node q in a TPQ Q, the element indicated by the cursor C_q is the head element of q.*

Definition 6.17 (Child and Descendant Extension)**.** *A query node q has the child and descendant extension if the following properties hold:*

- $\forall\, n_i \in childrenAD(q)$, *there is an element* e_i *which is the head of* T_{n_i} *and a descendant of* e_q *which is the head of* T_q.
- $\forall\, n_i \in childrenPC(q)$, *there is an element* e_q *which is the head of* T_q *and its CPL parameter is divisible by tagPrime*(n_i).
- $\forall\, n_i \in children(q)$, n_i *must have the child and descendant extension.*

The above definitions are fundamental to establish the correctness of the following lemmas:

Lemma 6.18. *For any arbitrary query node* q' *which is returned by getNext(q), the following properties hold:*

1. q' *has the child and descendant extension.*
2. *Either (a)* $q = q'$ *or (b)* q' *violates the child and descendant extension of the head element* e_q *of its parent*(q').

Proof. (Induction on the number of child and descendants of q). If q is a leaf query node, it is returned in Line 2 because it verifies all the properties 1 and 2a in Lemma 6.18. Otherwise, the algorithm recursively gets g_i = *getNext*(n_i) for each child of q in Line 4. If for some i, there is $g_i \neq n_i$, and it is known by inductive hypothesis that g_i verifies the properties 1 and 2b with respect to q, so the algorithm returns g_i in Line 6. Otherwise, by inductive hypothesis that all q's child nodes satisfy properties 1 and 2a with their corresponding sub-queries. At *getElement(q)* (Lines 21-25), *getNext* advances from T_q all segments that do not satisfy the divisibility by the product of prime numbers in $childrenPC(q)$ returned from *getQCPL*. After that, the algorithm advances from T_q (Lines 9-10) all segments that are beyond the maximum start value of $n_i \in children(q)$. Then, if q satisfies properties 1 and 2a, it is returned at Line 12. Otherwise, Line 13 guarantees that $n_i \in children(q)$ with the smallest start value satisfies properties 1 and 2b with respect to start value of q's head element e_q is returned. □

Lemma 6.19. *Let* $e_1, e_2, \ldots, e_m$ *be a sequence of elements corresponding to the same query node q and returned by getNext. Then* $getStart(e_1) < getStart(e_2) < \cdots < getStart(e_m)$

Proof. Query node q is either leaf or internal. If q is a leaf query node, by Lemma 6.18, q satisfies properties 1 and 2b since the parent of q denoted as p=parent(q) has start value greater than the start value of q, such that getStart(q) < getStart(p), *getNext* returns q at Line 13, therefore all elements in the stream of q are returned in ascending order of their start values as they are sorted in their corresponding stream from the definition of *tag streaming scheme*. Otherwise, all elements which are skipped at Lines 9-10 of *getNext* or Lines 21-25 of *getElement(q)* are guaranteed not to be part of any child and descendant extension. By Lemma 6.18, q is returned, then it satisfies properties 1 and 2a, thus all

elements in the stream of q are returned in ascending order of their start values as they are sorted in their corresponding stream *tag streaming scheme*. For both cases the lemma holds. □

The lemma above guarantees that when an element of a query node q is returned by *getNext* algorithm, there is no element remaining in the stream which has a start value lower than the start value of the element returned. This shows the difference between TwigStackPrime and TwigStackList with respect to the element processing order (see Section 4.2.3) since TwigStackList guarantees this property only for leaf query nodes while internal query nodes may be returned in an order violating the sorted order of the streams. Because of this, TwigStackList can not be used to accelerate one-phase holistic algorithms nor produce a query result conforming to the definition of TPQs (see Definition 4.11) [132, 89].

Lemma 6.20. *Suppose $getNext(q)$ returns a query node q' and $q == q'$. Then there is no further solution involving some elements of children(q') which have start values less than the start value of the head element of q'.*

Proof. Suppose that on contrary, there is a new solution using some already processed elements of q' in $S_{q'}$ denoted as $e_{S_{q'}}$ for which getEnd($e_{S_{q'}}$) < getStart(q'). Using range-based property, it will be inferred that all elements in the streams of children(q') must have end values less than getEnd($e_{S_{q'}}$), thus less than getStart(q'). Since $getNext(q) = q'$, it is known by Lemma 6.18 that q' has child and descendant extension, therefore all elements in the streams of children(q') must have start values larger than the start value of the head element of q', which is a contradiction. □

Lemma 6.21. *Suppose $getNext(q)$ returns a query node q' and $q \neq q'$ at either Line 4 or 13 of getNext. Then there is no new solution involving top element of the parent stack of q' denoted as p which has end value less than the start value of the head element of q' or some elements which are in children(p).*

Proof. Suppose that on contrary, there is a new solution using some elements of p = parent(q') in S_p denoted as e_{S_p} for which getEnd(e_{S_p}) < getStart(q'). Using range-based property, it will be known that all elements from children(p) in some solutions must have end values less than the end value of e_{S_p}, therefore less than the start value of the head element of q'. Since $getNext(q) = q'$ and from Line 3 of *getNext* for each child node n_i of p (including q'), it is *getNext(n_i)* = n_i and getStart(q') $\leq$ getStart(n_i). Using Lemma 6.18, it will be known that each n_i has a child and descendant extension, and thus all elements of children(n_i) have start values greater than getStart(n_i), therefore greater than getStart(q'), which is a contradiction. □

The above lemmas guarantee that all elements in the XML tree which are part of some solutions at subtree rooted at a query node in a TPQ will be returned in a document order and a stack for each query node is sufficient to keep track of all elements which may

contribute in further path solutions. Lemma 6.20 is very important to verify the use of the *getNext* with CPL in the family of one-phase holistic algorithms. Moreover, both prove also a chain of stacks can be used to represent partial solutions similar to TwigStack that is the chain of stacks represents paths containing the similar chain of elements as appear in the XML tree [40]. Finally, each time *getNext* returns a leaf query node, the corresponding root-to-leaf path can be outputted using the head element of the query node returned.

Lemma 6.22. *For any element e_q of a query node q in the stack S_q. If e_q is popped from S_q, then it does not contribute to any further solution.*

Proof. In TwigStackPrime, any element is popped off the stack by either the head element of $n_i \in children(q)$ or the head element of q. In the case of the head element which corresponds to children(q), the proof can be obtained using Lemma 6.21. In the case of the head element has the same query node, the proof of this by using Lemma 6.20. □

The above lemma is very important to ensure that each time *getNext* returns a query node q and the head element of q does not have the ancestor extension, the algorithm can safely pop out elements in the parent stack since they do not contribute in any new solution. If the stack is empty, the current head element of q can be also skipped and the cursor of q can be advanced to the next element in the stream of q. Using the above lemmas, the next theorem will be used to prove the correctness of TwigStackPrime and its core function *getNext*.

Theorem 6.23. *Given a twig pattern query Q and an XML document D, Algorithm TwigStackPrime correctly returns the answer to Q on D.*

Proof. In Algorithm *TwigStackPrime*, *getNext(root)* is repeatedly invoked to determine the next query node to be processed. Using lemma 6.18, it is known that all elements returned by $q_{act} = getNext(root)$ have the child and descendant extension. If $q_{act} \neq root$, Line 4, the algorithm pops from $S_{parent(q_{act})}$ all elements that are not ancestors of the head element of q_{act} by Lemma 6.21. After that, it is already known q_{act} has a child and descendant extension so that Line 5 checks whether $S_{parent(q_{act})}$ is empty or not. If so, it indicates that it does not have the ancestor extension, and it can be discarded safely to continue with the next iteration. Otherwise, the current head element of q_{act} has both the ancestor and child and descendant extensions which guarantee its participation in at least one root-to-leaf path. Then, $S_{q_{act}}$ is cleaned by popping elements which do not contain the head of q_{act}, using Lemma 6.22. Then, the item in the stack is used to maintain pointers from itself to the query root. Finally, if q_{act} is a leaf query node, all possible combinations of single paths with respect to q_{act} can be computed at Lines 8-9 and stored in the corresponding output array. □

The correctness holds for TPQs with both Ancestor-Descendant and Parent-Child relationships, it can be inferred that TwigStackPrime is optimal for the case where the TPQ has Ancestor-Descendant edges or there are only Parent-Child edges connected the

leaf query nodes. The intuition is simple since The CPL relationship can detect hidden immediate child elements only in two streams related by P-C relationship due to the restricted access mechanism. Thus, elements are only stored in the intermediate result if they have child and descendant extension. Therefore, the merge postprocessing operation is optimal. However, in the case where P-C axes connects internal query nodes, the algorithm can not guarantee that the inferred, hidden elements in the streams have child and descendant extensions by using Lemma 6.18 since they have not been seen as head elements yet. As a result, the following result can be concluded.

Theorem 6.24. *Consider a twig pattern query Q with n query nodes, and only Ancestor-Descendant edges or there are Parent-Child edges to connect leaf query nodes, and an XML document D. TwigStackPrime has worst-case I/O and CPU time complexity linear in the sum of the size of the n input lists and the output list.*

Since TwigStackPrime maintains stacks to encode individual root-to-leaf paths, the worst-case size of any stack in TwigStackPrime is equal the maximum depth in the XML tree. Due to this fact the following result can be obtained.

Theorem 6.25. *Consider a twig pattern query Q with n query nodes, and an XML document D. The worst-case space complexity of TwigStackPrime is proportional to the longest path in D times n.*

The following section describes the experiments to evaluate the performance of the algorithm proposed and to test the research hypothesis introduced in Section 4.3.2. It also presents the experimental results on the performance of three top-down holistic twig matching algorithm.

6.5 Experimental Evaluation

The main objective of the experiments in this section is to compare the performance and scalability of TwigStackPrime against other top-down holistic algorithms in the literature. Moreover, the aim of this section is to present the experimental results of the performance comparison of top-down twig join algorithms, namely: TwigStackPrime the new algorithm, proposed in this thesis, based on *Child Prime Labels*, along with TwigStack [40], which is the original twig join algorithm, that was reported to have worst case I/O and CPU time complexity which are linear with respect to the TPQ result size when the TPQ has only A-D relationships in all edges, and TwigStackList is the first refined version of TwigStack to handle P-C edges efficiently [144]. TwigStackList was chosen in this experiment because it utilizes a simple buffering technique to prune irrelevant elements from the streams. TwigStackList was reported to allow a TPQ processing with a linear I/O and CPU time complexity with respect to the TPQ result size for a TPQ with A-D relationships in all axes or P-C edges which exist only in non-branching edges. Recall that from the theoretical

analysis in the above section, TwigStackPrime can guarantee optimal evaluation for a TPQ with respect to the TPQ result size for a TPQ containing either A-D relationships in all edges or the only P-C edges connect leaf query nodes with their parents. The research hypothesis, which was stated in Chapter 4, will be tested using the evaluation process.

6.5.1 Experimental Setting

All the algorithms tested in the experiment were implemented and added to the query processor discussed in Section 5.2.2.2. The experiments were executed in the platform described in Section 5.3. Since the intermediate results are stored in memory, output arrays are implemented using the ArrayList implementation in Java. To ensure comparability, for each algorithm the ArrayLists used are tuned and initialised using the capacity parameter with the value of the maximum number of single paths produced by the algorithm for the purpose of eliminating resizing costs [215]. Although this provision comes at the expense of memory consumption, it is believed that such a cost may affect the running time of the algorithm and because of the TwigStack algorithm does not apply a sophisticated mechanism to control the number of intermediate paths for TPQs with P-C edges, such a cost may be counted as a speed-up mechanism for the other algorithms [202, 156]. Also note that outputting path solutions to the secondary storage may lead to a high number of random disk access with the TwigStack algorithm as was reported in the experiments of [21, 116].

6.5.1.1 XML Datasets and Queries

The datasets selected are those most frequently used in the literature of XML query processing [40, 144, 89, 237, 132, 185]. Four datasets were chosen out of the five datasets utilised in this thesis because the Zipf dataset was designed to investigate the performance of the one-phase family of holistic algorithms (see Section 5.4.3.2). As a result, the DBLP (see Section 5.4.1.1), XMark with the scaling factor equal to 1 (see Section 5.4.2.1), TreeBank (see Section 5.4.1.2) and Random (see Section 5.4.3.1) datasets were used in the experiments. Table 6.4 presents statistical information about the five datasets labelled with the original range-based labelling scheme and a combination of range-based and the CPL index. The XML structured queries for evaluation over these datasets were chosen specifically to cover the classes of TPQ which fall within or outside the optimal groups of the comparable algorithms, to put it another way, they include members of the class that are optimal as well as members of the class that are not optimal. Furthermore, the queries chosen have different structures and distributions of A-D and P-C edges. The queries over DBLP, XMark and TreeBank datasets were previously used in the literature, they were obtained from several existing studies on TPQ processing [40, 144, 22, 89, 54, 18, 188], while TPQs over the Random dataset were created by replacing randomly the tag names for the existing queries with the labels of the Random dataset. They were grouped into

Table 6.4: Datasets statistics

	DBLP	XMark	TreeBank	Random	Zipf
Rangae-based MB	65.3	35.3	43	69.4	68.33
Rangae-based + CPL MB	70.3	40.1	47.9	74.1	74
$\triangle$ size MB	5	4.8	4.9	4.7	5.76
Tag Indexing Size KB	0.48	1	3	0.049	0.056
Distinct Tags	40	83	251	6	7
Largest CPLs generated	$6.3842417e^{+18}$	$3.052156e^{+15}$	$9.123407e^{+18}$	30,030	30,030

Table 6.5: Experimental TPQs for DBLP.

Code	Category	XPath expression	Result size
DQ_1	B	/dblp/inproceedings[//title]//author	88
DQ_2	C	//www[editor]/url	21
DQ_3	A	//article[//sup]//title//sub	278
DQ_4	-	//article[/sup]//title/sub	0

four categories: category A for A-D axes, category B for P-C edges in non-branching axes and category C for P-C edges related to leaf query nodes. Recall that TwigStack is optimal for A and TwigStackList is optimal for both A and B, while TwigStackPrime is optimal for A and C. Category "-" for other classes of queries. For the sake of simplicity, they were also encoded, the code indicates a combination of dataset and its TPQ. By way of illustration, TQ2 refers to the second TPQ issued over TreeBank dataset. The properties of TPQs selected over DBLP are given in Table 6.4. The characteristics of the queries over XMark are presented in Table 6.6. Tables 6.7 and 6.8 provide an overview of the TPQs over TreeBank and Random datasets, respectively.

6.5.1.2 Metrics

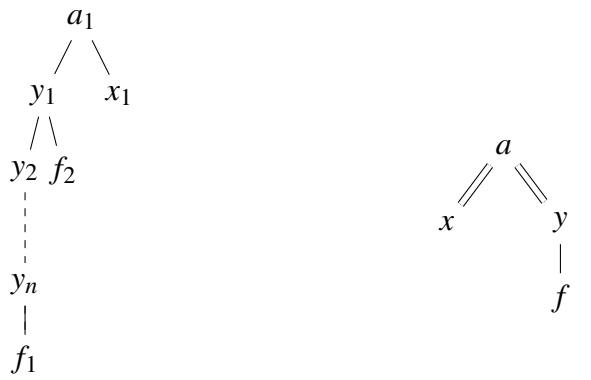
The experiments compare two variables for each TPQ selected under the three top-down holistic algorithms. Consequently, the performance comparison of these algorithms was based on the following metrics:

- Number of intermediate path solutions: the individual root-to-leaf paths generated by each algorithm and stored in the output arrays.

- Processing time: the running time of an algorithm spent on the whole TPQ includes both phases (in milliseconds). All TPQs were executed 103 times to increase the reliability of measures and the first three runs were excluded for cold cache issues.

Table 6.6: Experimental TPQs for XMark.

Code	Category	XPath expression	Result size
XQ_1	-	/site/closed_auctions/closed_auction[annotation/description/text/keyword]/date	4042
XQ_2	A,B and C	/site/closed_auctions/closed_auction//keyword	12527
XQ_3	-	/site/closed_auctions/closed_auction[//keyword]/date	12527
XQ_4	-	/site/people/person[profile[gender][age]]/name	3243
XQ_5	-	//item[location][//mailbox//mail//emph]/description//keyword	16956
XQ_6	-	//people/person[//address/zipcode]/profile/education	3241



(a) an XML tree T_3.

(b) a twig query Q_1.

Figure 6.9: Illustration to the difference between TwigStackList [144] and TwigStackPrime.

Table 6.7: Experimental TPQs for TreeBank.

Code	Category	XPath expression	Result size
TQ_1	A	//S[//MD]//ADJ	19
TQ_2	-	//S/VP/PP[/NP/VBN]/IN	152
TQ_3	C	//VP[/DT]//PRP_DOLLAR_	3
TQ_4	C	//S[/JJ]/NP	5
TQ_5	-	//S[VP[DT]//NN]/NP	32
TQ_6	B and C	//S[//VP/IN]//NP	20311
TQ_7	-	//S/VP/PP[//NP/VBN]/IN	320
TQ_8	-	//EMPTY/S//NP[/SBAR/WHNP/PP//NN]/_COMMA_	17
TQ_9	-	//SINV//NP[/PP//JJR][//S]/NN	4
TQ_{10}	C	//NP[/NN]/PP	43942

Table 6.8: Experimental TPQs for Random.

Code	Category	XPath expression	Result size
RQ_1	C	//b//e//a[//f][d]	1331
RQ_2	C	//a//b[//e][c]	18033
RQ_3	C	//e//a[/b][c]	11216
RQ_4	B and C	//a[//b/d]//c	59568
RQ_5	-	//b[d/f]/c[e]/a	377
RQ_6	-	//c[//b][a]/f	47159
RQ_7	-	//a[c//e]/f[d]	1906
RQ_8	-	//d[a//e/f]/c[b]	204
RQ_9	C	//a[d][c][b][e]//f	3757

The I/O cost for tag indexing files for *TwigStackPrime* algorithm is not counted because it is negligible, and the cost to read the tag indexing is constant over a series of twig pattern queries for each dataset. In other words, it needs only to be read once for a set of TPQs over a particular dataset.

6.5.2 Experimental Results

This section describes the evaluation of the experimental results. In the first place, all the results returned from the algorithms were inspected, and were found to be all the same which verifies the validity of the new approach. To allow precise comparisons, the discussion of the query performance related to a particular dataset is contained within an individual subsection. The query performances for TPQs over DBLP, XMark, TreeBank and Random datasets are evaluated in Sections 6.5.2.1, 6.5.2.2, 6.5.2.3 and 6.5.2.4, respectively. The scalability tests are discussed in Section 6.5.2.5.

6.5.2.1 DBLP

This section shows the experimental results for TPQs over the DBLP dataset, the TPQs are given in Table 6.5. This real world dataset is wide and shallow so that it is not common for TPQs, which contain both '//' and '/', to make a significant difference in performance for tightly-structured XML documents. Figure 6.10 depicts the number of intermediate results generated by each algorithm along with the actual path solutions participating in the query results.

Even though most of the TPQs tested do not fall within the optimal sets for TwigStackList and TwigStackPrime algorithms, they both show optimal processing for all TPQs over this dataset, while TwigStack generates useless paths in DQ_1 and DQ_4. Note that DQ_4 has no match in the dataset. Figure 6.11 shows the performance of the algorithms, note that the query processing time in seconds is to enable a clear representation of wide diversity using the same plot. As can be seen from the experimental results, TwigStackPrime has the best performance for most the queries. To compare statistically the overall query performance, the Kruskal-Wallis test for each TPQ was carried out to see whether there was a difference

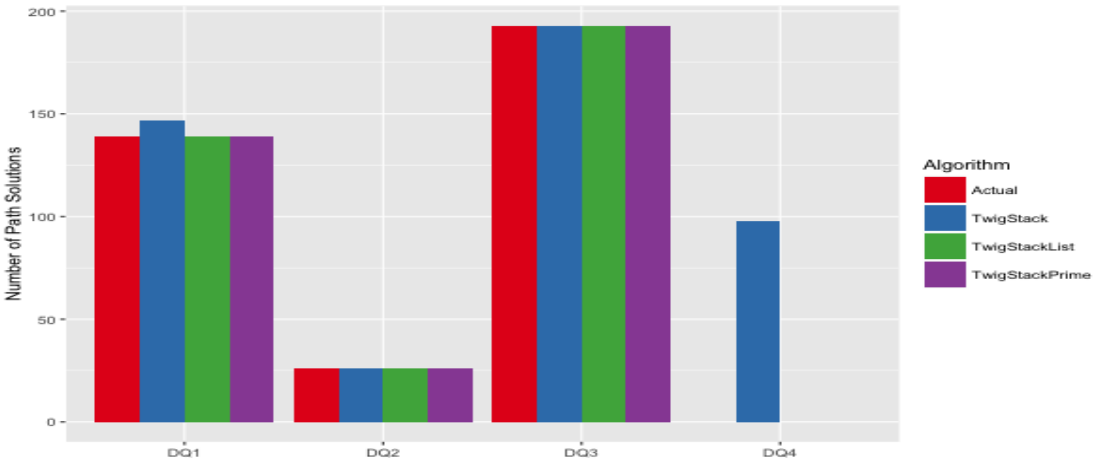


Figure 6.10: The number of intermediate single paths generated by each algorithm for the queries tested over DBLP. "Actual" represents the number of path solutions relevant to the query results.

Table 6.9: Results for the comparison groups over DBLP.

Query	p-value	p-value < 0.05
DQ1	2.28E-57	TRUE
DQ2	4.66E-21	TRUE
DQ3	9.61E-40	TRUE
DQ4	1.90E-46	TRUE

in the performance between at least two algorithms or not. Table 6.9 presents the results of running the Kruskal-Wallis tests over the queries using the significance level *0.05* to test the null hypothesis. Consequently, it can be seen from the table that the Kruskal-Wallis tests suggest that there is a significant difference in the performance between two algorithms at least. Therefore, the Mann-Whitney U test was run for all possible combinations of pair over the queries tested. For K groups (i.e., the algorithms) and T the number of TPQs which reject the null hypothesis of the Kruskal-Wallis test, the number of pairwise comparisons can be expressed using the following formula:

$$\#\ of\ U\ tests = \frac{(k \times (k-1))}{2} \times T \tag{6.2}$$

Using the formula in 6.2, the number of paired comparisons for this dataset is calculated as $= \frac{(3\times(3-1))}{2} \times 4 = 12$. The complete results for the paired comparisons can be found in Appendix A. The overall results are presented in Table 6.10 which summarises the comparisons to show how many times each algorithm statistically outperformed or was outperformed by the algorithm it was compared with.

As shown in Table 6.10, TwigStackPrime outperformed the other algorithms in all TPQs except for DQ_3, a possible explanation is that DQ_3 falls within the A category where TwigStack is optimal and TwigStackPrime introduces some run-time overhead in order to improve its optimality features. Interestingly, TwigStackList and TwigStackPrime are

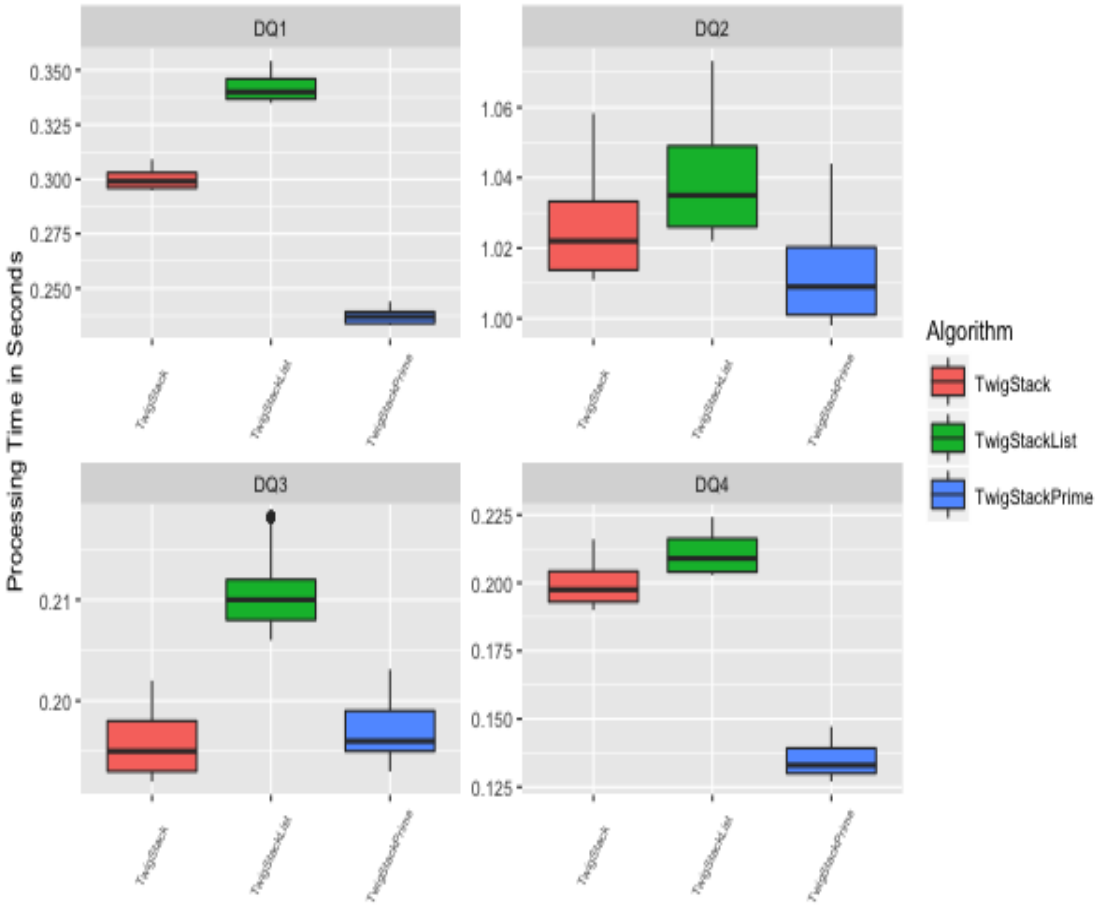


Figure 6.11: Query processing time of the algorithms compared for TPQs against DBLP.

Table 6.10: The overall comparisons based on U tests over DBLP. "-" indicates no difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
TwigStack	5	3	0
TwigStackList	0	8	0
TwigStackPrime	7	1	0

extensions of TwigStack but it can be observed that TwigStackPrime has significantly better performance than TwigStackList for DQ_3, this could be expected because it does not need to perform extra investigations with the absence of P-C edges as TwigStackList has to check its lists every time a new element is scanned and skipped. For DQ_3, the query processing time difference between TwigStack and TwigStackPrime is 1 millisecond and the effect size is very small r= 0.18, while the difference with TwigStackList is around 15 milliseconds with a large effect size r = 0.86. It should be noted that the effect size tests suggested moderate to high practical significance with the sole exception of DQ_3 between TwigStack and TwigStackPrime.

To conclude, the number of intermediate results and query processing time for TwigStackPrime were better than the other algorithms in most cases. It can be observed that for

TPQs with A-D edges, both TwigStack and TwigstackPrime have very similar performance. However, the number of intermediate path solutions produced by TwigStackList and TwigStackPrime are the same due to the fact that DBLP does not have recursive tags.

6.5.2.2 XMark

In the XMark dataset, the experiment is to test the performance of the algorithms on a relatively balanced XML tree. The number of intermediate results are shown in Figure 6.12. Similar to the experimental results obtained from DBLP, both TwigStackList and TwigStackPrime showed optimal performance regardless the classes of TPQs, whereas TwigStack can not perform efficiently for XQ_1. This can be explained that XMark dataset conforms to a predefined scheme so that the query matches have relatively uniform distribution. Figure 6.13 depicts query processing overall performance for this experiment.

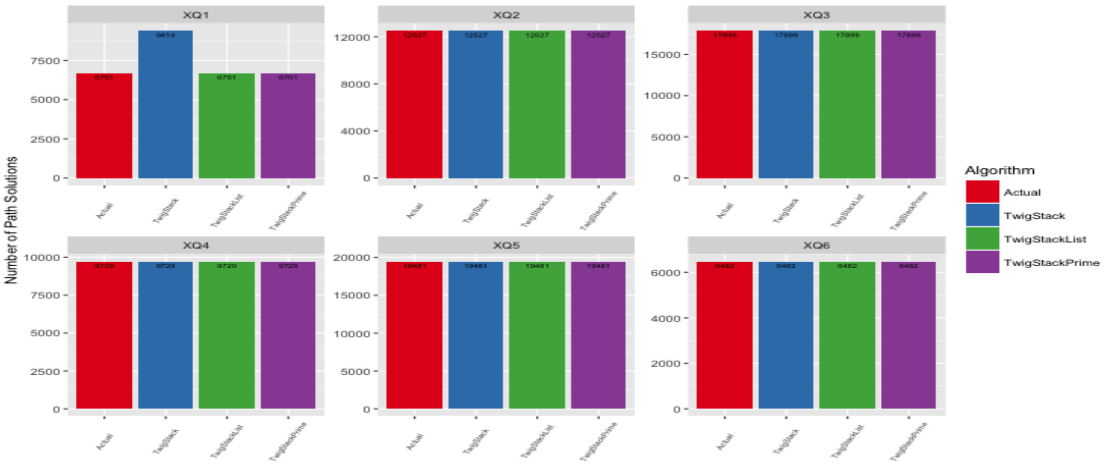


Figure 6.12: The number of intermediate single paths generated by each algorithm for the queries tested over XMark. "Actual" represents the number of paths appearing in the final matches.

To compare the query performance, the experiment hypothesises that there is no difference in the performance between the algorithms so that the Kruskal-Wallis test was carried out to test that null hypothesis. Table 6.11 presents an overview of group comparisons based on Kruskal-Wallis tests. Since Kruskal-Wallis tests suggested that there is a significant difference in the performance between two algorithms at least for all TPQs, the total number of paired comparisons using Formula 6.2 is obtained as follows: $= \frac{(3\times(3-1))}{2} \times 6 \ = 18.$

Combining the figures from Table 6.12 and the experimental results given in Figure 6.13, TwigStackPrime showed the best performance in most cases, it performed slightly slower than TwigStack in two TPQs, namely $XQ2$ and XQ_6. All the algorithms are optimal with respect to XQ_2 because it is a non-predicate query (i.e., simple path expression). The reason for selecting this query for the experiments was to test the performance of the algorithms when there are no branching edges. Both TwigStackList and TwigStackPrime have a similar performance with only 1 millisecond slower than TwigStack. When comparing

Table 6.11: Results for the comparison groups over XMark dataset.

Query	p-value	p-value < 0.05
XQ1	4.93E-58	TRUE
XQ2	1.67E-09	TRUE
XQ3	2.01E-23	TRUE
XQ4	3.07E-40	TRUE
XQ5	8.19E-43	TRUE
XQ6	2.28E-14	TRUE

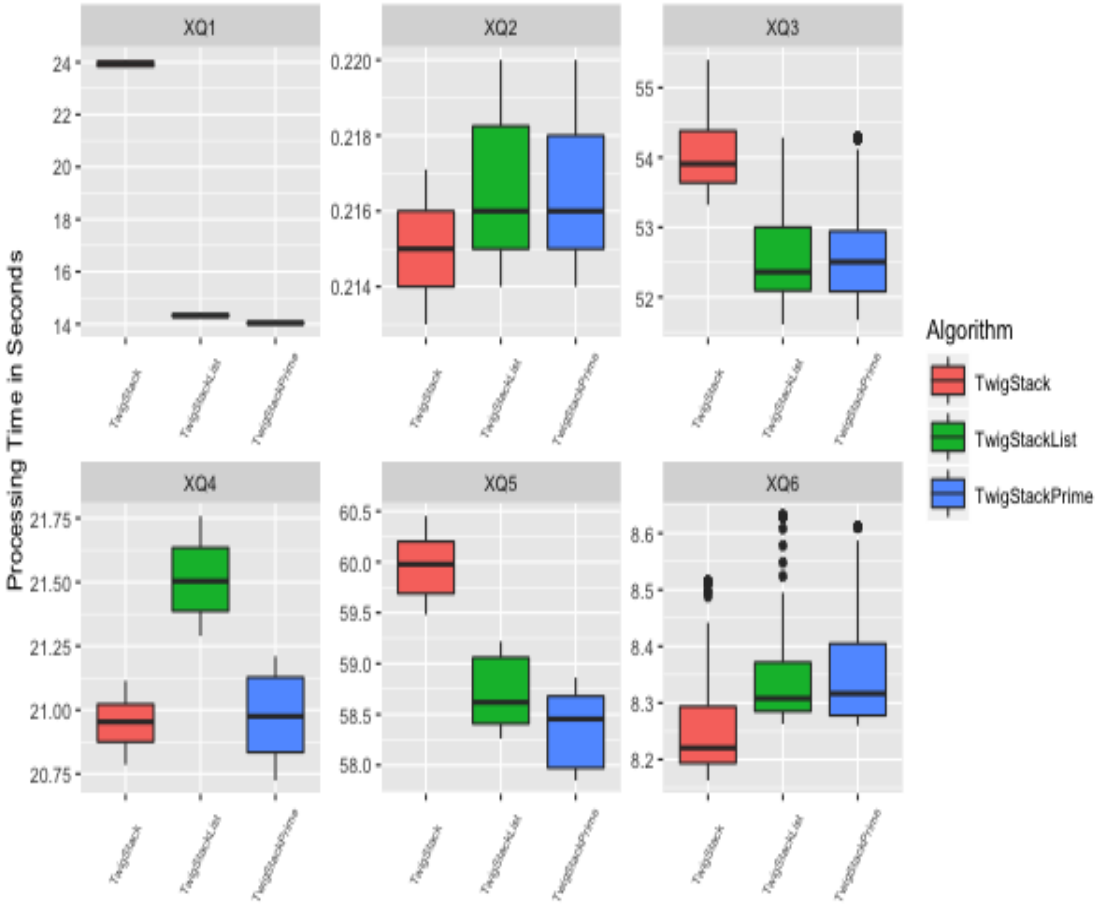


Figure 6.13: Query processing time of the algorithms compared for TPQs against XMark.

Table 6.12: The overall comparisons based on U tests over XMark dataset. "-" indicates no difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
TwigStack	5	6	1
TwigStackList	3	6	3
TwigStackPrime	6	2	4

TwigStackPrime with TwigStackList, TwigStackPrime outperformed the latter in three queries, namely XQ_1,$XQ4$ and XQ_5. However, they performed the same for the rest of the

TPQs. With regard to the effect size tests, most of the results suggested medium to large practical significance. The raw data for the paired comparisons is presented Appendix A.

To conclude, for this type of dataset, TwigStackPrime has a superior performance to the other techniques in terms of the number of paths generated and query running time. When subtrees in the dataset have unbalanced structures such as the one examined by XQ_1, TwigStack can not prevent the generation of useless paths. For XQ_1, TwigStackPrime ran two times faster than TwigStack and provided optimal processing.

6.5.2.3 TreeBank

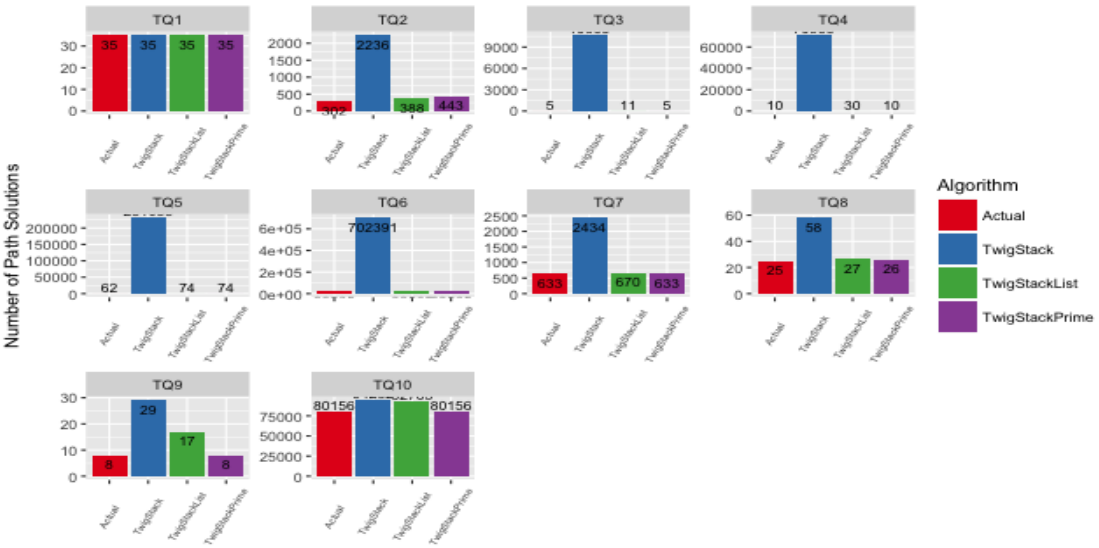


Figure 6.14: The number of intermediate single paths generated by each algorithm for the queries tested over TreeBank. "Actual" represents the number of paths contributing in the final matches.

Ten TPQs (see Table 6.7) were issued over TreeBank to test the performance of the algorithms where the dataset has an extremely irregular structure. Figure 6.14 shows the number of path solutions produced by each algorithm. Unlike the previous datasets, the TreeBank dataset is deeply recursion so that the suboptimal evaluation of TwigStack can be demonstrated here as depicted in the illustrative graph. Due to the gap between TwigStack and the other algorithms, each TPQ is depicted in an individual plot for the sake of clear presentation. For example, 5 actual path solutions are the only participants to find matches to TQ_3, the number of simple paths produced by each algorithm is as follow: TwigStack and TwigStackList generated 10,663 and 11 paths, respectively, while TwigStackPrime produced 5 paths. It can be seen that TwigStackPrime is the only algorithm which can process TQ_3 and TQ_4 efficiently in terms of the number of path solutions because they fall within the optimal class of TPQs where TwigStackPrime guarantees optimal processing according to Theorem 6.24. However, in TQ_2, no algorithm can guarantee optimal evaluation, the number of useful path solutions is 302 but the algorithms TwigStack, TwigStackList and TwigStackPrime produced 2,236, 388 and 443 paths, respectively. For

Table 6.13: Results for the comparison groups over TreeBank dataset.

Query	p-value	p-value < 0.05
TQ1	9.69E-49	TRUE
TQ2	2.87E-58	TRUE
TQ3	8.92E-59	TRUE
TQ4	5.42E-54	TRUE
TQ5	1.93E-58	TRUE
TQ6	1.05E-56	TRUE
TQ7	1.86E-58	TRUE
TQ8	1.88E-58	TRUE
TQ9	1.86E-58	TRUE
TQ10	1.18E-65	TRUE

TQ_6, TwigStackList and TwigStackPrime performed efficiently by generating only useful paths of 22,565, whereas the number of intermediate paths in TwigStack was 702,391. Figure 6.15 shows the execution time of the algorithms over this dataset. In order to the group comparisons, ten Kruskal-Wallis tests were carried out over TPQs to see whether there was a difference in the performance between the algorithms or not. All the Kruskal-Wallis tests suggested that there is a difference in the performance between two algorithms at least as presented in Table 6.13. Consequently, the paired comparisons based on the U test of Mann Whitney were calculated. The number of paired comparisons for this dataset can be obtained using Formula 6.2 as $= \frac{(3\times(3-1))}{2} \times 10 \ = 30$. The full results of the pairwise comparisons can be found in Appendix A.

Even though the query TQ_1 has only A-D edges, TwigStackPrime was faster than TwigStackList and slower than TwigStack by only 1 millisecond. The effect size test suggested that this had a large practical significance with TwigStackList and a moderate practical significance with TwigStack. As observed, TwigStackPrime does not sacrifice the performance when a TPQ has all A-D axes as it is the optimal class of TwigStack. The illustrative graph in Figure 6.15 shows that the only TPQs where TwigStackPrime was slower compared to the others are TQ_3 and TQ_9 because they touch very little of the dataset. In contrast, TwigStackPrime showed a superior performance in avoiding the storage of unnecessary paths while the cost difference between the compared algorithms is insignificant.

TQ_6 is a very expensive query, it touches a very large portion of the document and has a great deal of results. The Pairwise comparison based on Manny-Whitney test between TwigStackPrime and TwigStackList resulted in $p - value < .001$ which suggests a significant difference. TwigStackPrime had the best performance and the effect size suggested a large practical significance as TwigStackList was 610 milliseconds slower than TwigStackPrime. Table 6.14 provides the interpretation of the results returned from the paired comparisons between the algorithms.

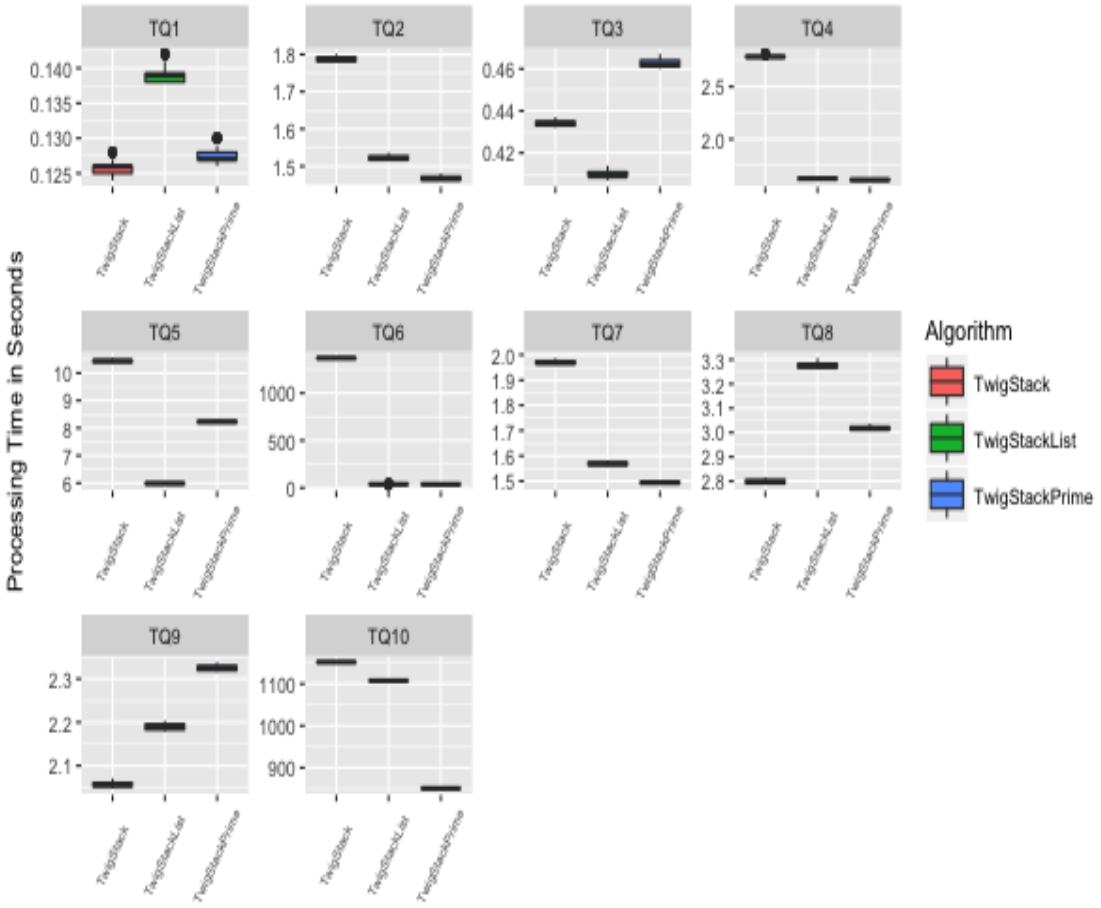


Figure 6.15: Query processing time of the algorithms compared for TPQs against TreeBank.

Table 6.14: The overall comparisons based on U tests for TreeBank dataset. "-" indicates no difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
TwigStack	7	13	0
TwigStackList	10	10	0
TwigStackPrime	13	7	0

To sum up, TwigStackPrime has shown a superior performance to the other algorithms in terms of the number of intermediate results and query running time for deeply recursive dataset. For TPQs in category C, TwigStackPrime did not produce useless paths, whereas the others did. This verifies the analysis of its optimal sets of TPQs as stated in Theorem 6.24. The illustrative graph in Figure 6.15 shows that TwigStackPrime was faster than TwigStack and TwigStackList in 5 and 6 TPQs, respectively.

6.5.2.4 Random

The Random dataset has a complex structure with six distinct tags. This dataset was created to test the performance where the XML combines features of DBLP and TreeBank, being

relatively structured and deeply-recursive at the same time. The number of intermediate path solutions produced by each algorithm is presented in Figure 6.16

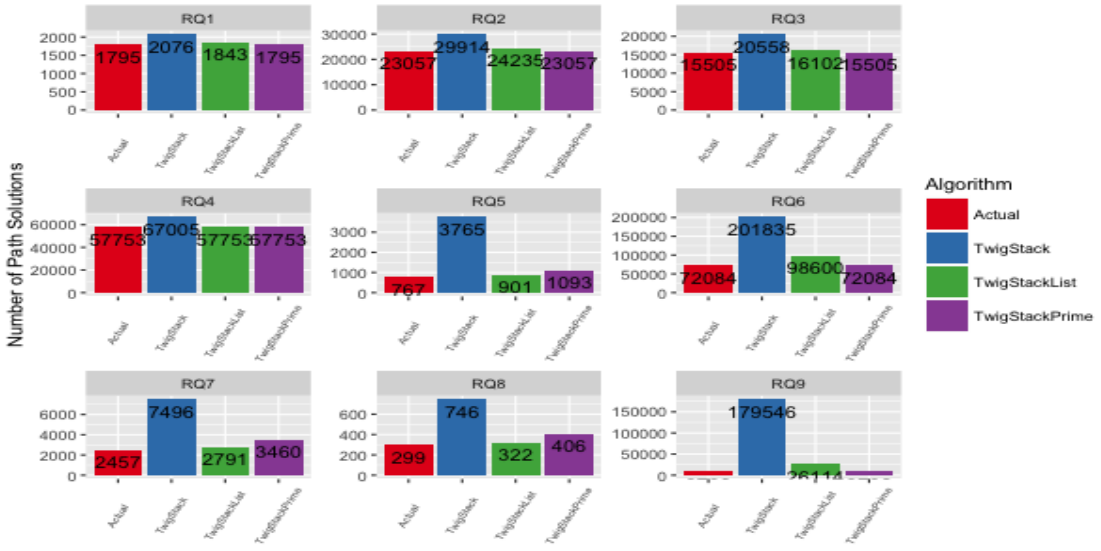


Figure 6.16: The number of intermediate single paths generated by each algorithm for the queries tested over Random. "Actual" represents the number of paths contributing in the final matches.

Five queries, namely RQ_1, RQ_2, RQ_3, RQ_6 and RQ_9, can be processed efficiently in terms of the number of intermediate single paths generated only by TwigStackPrime as shown by Figure 6.16. TwigStackList and TwigStackPrime are optimal for RQ_4, while TwigStack produced useless paths. Moreover, TwigStackPrime has shown the best evaluation for RQ_6 even though this query does not fall within its optimal sets of TPQs. TwigStackPrime produced 72,084 paths, while the number of intermediate paths in TwigStack and TwigStackList are 201,835 and 98,600 paths, respectively. Figure 6.17 illustrates the query processing time of the algorithms.

To compare the query performance, Kruskal-Wallis test was carried out to test the null hypothesis for each TPQ tested. The result of group comparisons based on Kruskal-Wallis test are summarised in Table 6.15. Since the results turned out to reject the null hypothesis by suggesting that for each TPQ there was a difference in the performance between two algorithms at least, all the possible paired comparisons were computed using Formula 6.2 as follows: $= \frac{(3\times(3-1))}{2} \times 9 = 27$. The full results detail are shown in Appendix A.

The summary of the paired comparisons based on the Mann Whitney U test is given in Table 6.16. As reflected in the table, TwigStackPrime was faster than the other algorithms in most cases, it had a similar performance with TwigStackList for RQ_4. For example, to evaluate RQ_6, TwigStackPrime had the best performance, it was roughly twice as fast than TwigStackList and five time faster than TwigStack, see Figure 6.17.

To conclude, TwigStackPrime outperformed TwigStack in 6 TPQs and was faster than TwigStackList in 5 TPQs. According to the experimental results, the presence of P-C edges in the lowest level of the individual paths of TPQs demonstrated the effect of

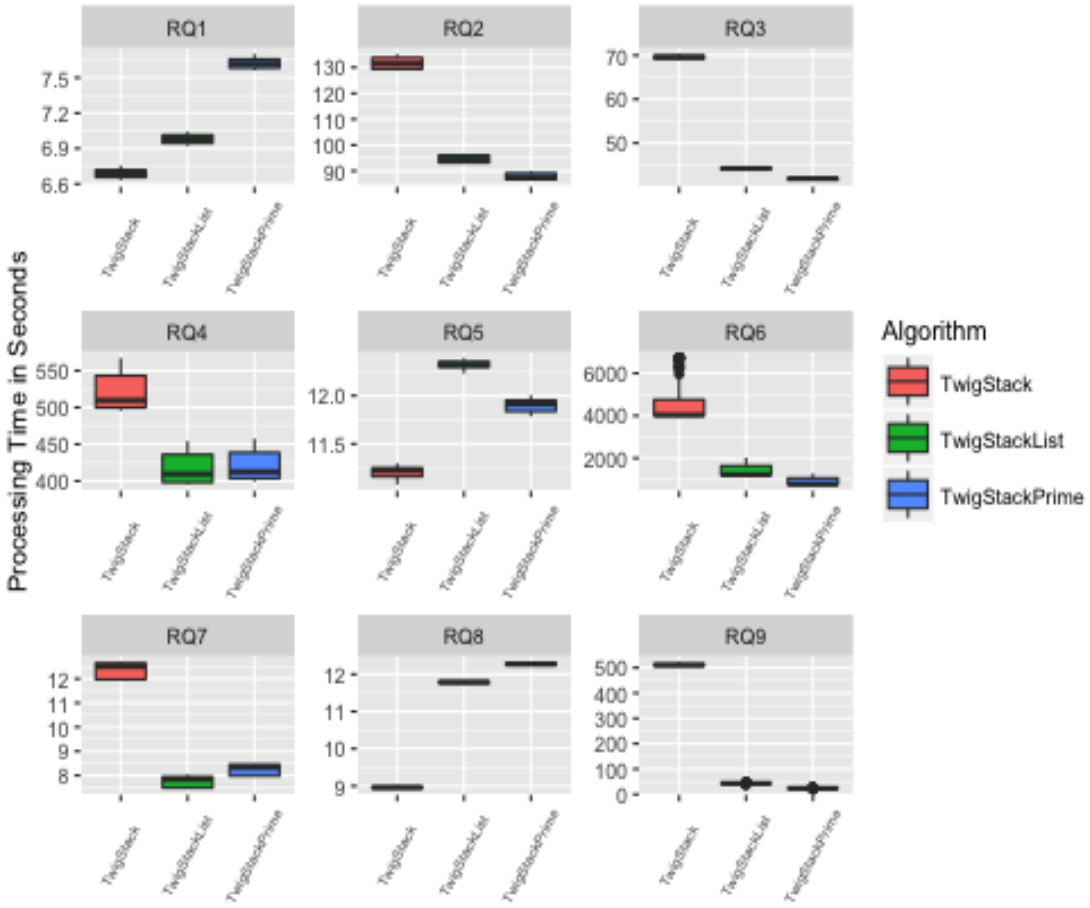


Figure 6.17: Query processing time of the algorithms compared for TPQs against Random.

Table 6.15: Results for the group comparisons over Random dataset.

Query	p-value	p-value < 0.05
RQ1	1.93E-58	TRUE
RQ2	1.93E-58	TRUE
RQ3	1.93E-58	TRUE
RQ4	2.38E-44	TRUE
RQ5	2.72E-58	TRUE
RQ6	7.09E-53	TRUE
RQ7	1.17E-54	TRUE
RQ8	1.92E-58	TRUE
RQ9	1.93E-58	TRUE

TwigStackPrime in providing efficient processing. Even the existence of P-C axes at the top level of TPQs, TwigStackPrime was more efficient than the other techniques. However, TwigStack had better performance than TwigStackPrime for TPQs with a relatively small number of intermediate results as an extra operation incurred by TwigStackPrime to eliminate irrelevant elements, namely RQ_1, RQ_5 and RQ_8. This can be explained that these TPQs have few occurrences in the dataset as presented in Table 6.8.

Table 6.16: The overall comparisons based on U tests over Random dataset. "-" indicates no difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
TwigStack	6	12	0
TwigStackList	9	8	1
TwigStackPrime	11	6	1

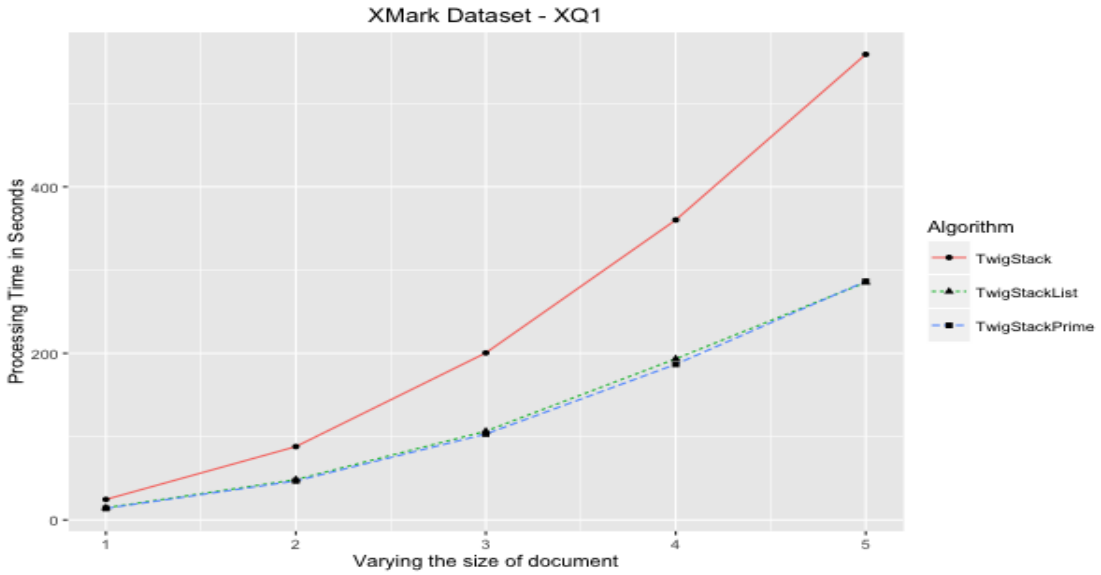
6.5.2.5 Scalability

This section aims to simulate and test the scalability of the new approach. In this experiment, two datasets were used, XMark and Random datasets. Whereas the XMark dataset is shallow and data oriented, the random collection has a very recursive structure. Five different versions of XMark were created using the scaling factor from 1 to 5 as explained in Section 5.4.4. The Random dataset was partitioned into 10 different datasets to evaluate the scalability of the algorithms over irregular datasets, see Section 5.4.4. In order to make the experiment more objective, two TPQs have been selected over each category of datasets, one of them was processed efficiently by TwigStackPrime in the query performance study while TwigStackPrime had the worst performance for the other one. As a result, XQ_1 and XQ_6 were selected for the XMark datasets, and RQ_8 and RQ_9 were chosen to be issued over the Random datasets. The results for XQ_1 and XQ_6 are illustrated in Figure 6.18. Since TwigStack produced useless paths for XQ_1, it did not scale well for this query. It can be observed that TwigStackList and TwigStackPrime scaled linearly with the increasing size of the dataset for XQ_1. On the other hand, for XQ_6, all the algorithms showed the same performance, they scaled almost linearly with the increasing size of the dataset.

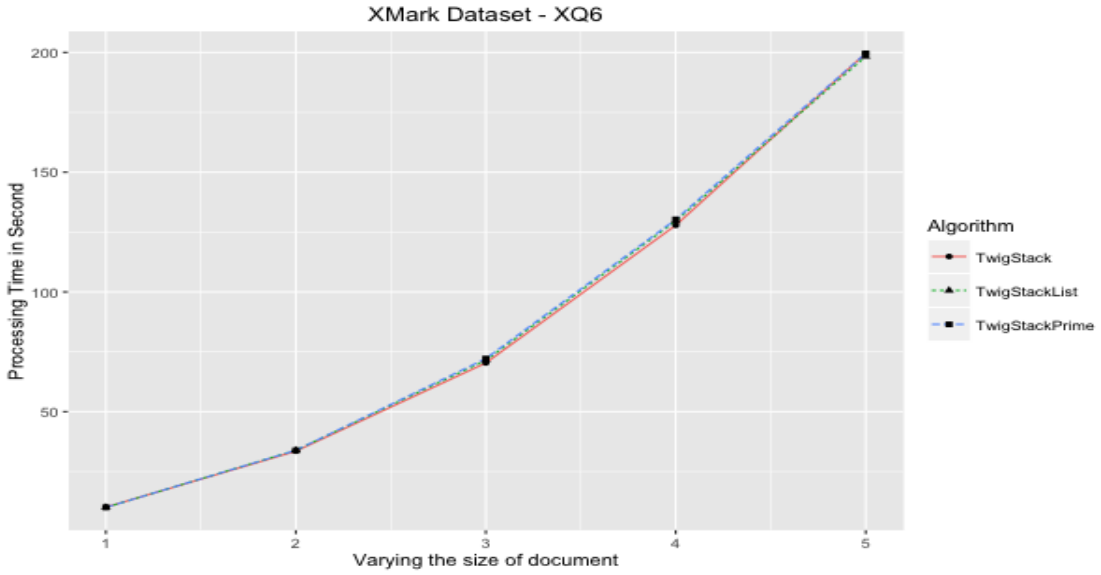
For Random datasets, the scalability results for RQ_8 and RQ_9 are depicted in Figure 6.19. Both TwigStackList and TwigStackPrime scaled effectively for RQ_8 even though they produced useless paths. In contrast, TwigStack started to increase slightly with the large datasets compared to the others. This can be explained because increasing the size of the dataset increases the number of useless paths for TwigStack, hence the processing time is slower as illustrated in Figure 6.19a. It can be seen from Figure 6.19b that the performance of TwigStackList and TwigStackPrime presented a linear relationship with the increasing of the size of the dataset. However, TwigStack had the worst scalability for RQ_9. Since the experiment includes variety of TPQs with different structures, it can be concluded that TwigStackPrime is more scalable in processing large, scale datasets.

6.5.3 Summary

The experimental results have shown that TwigStackPrime can filter out many irrelevant elements effectively and it can be observed that the number of path solutions produced by TwigStackPrime is usually significantly less than that generated by TwigStack. Furthermore, TwigStackPrime has the best performance in comparison to the comparable

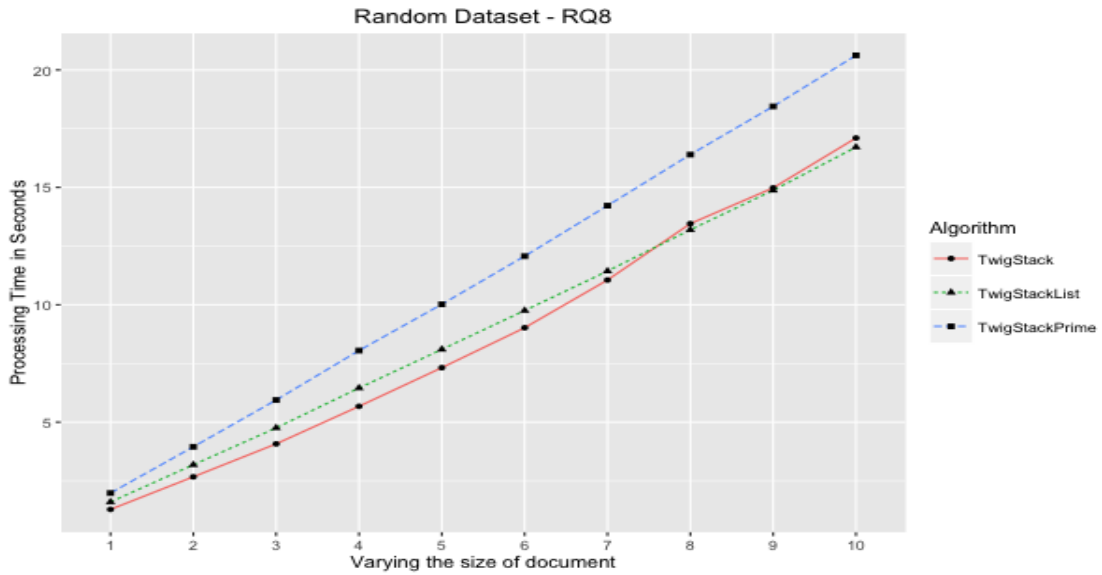


(a)

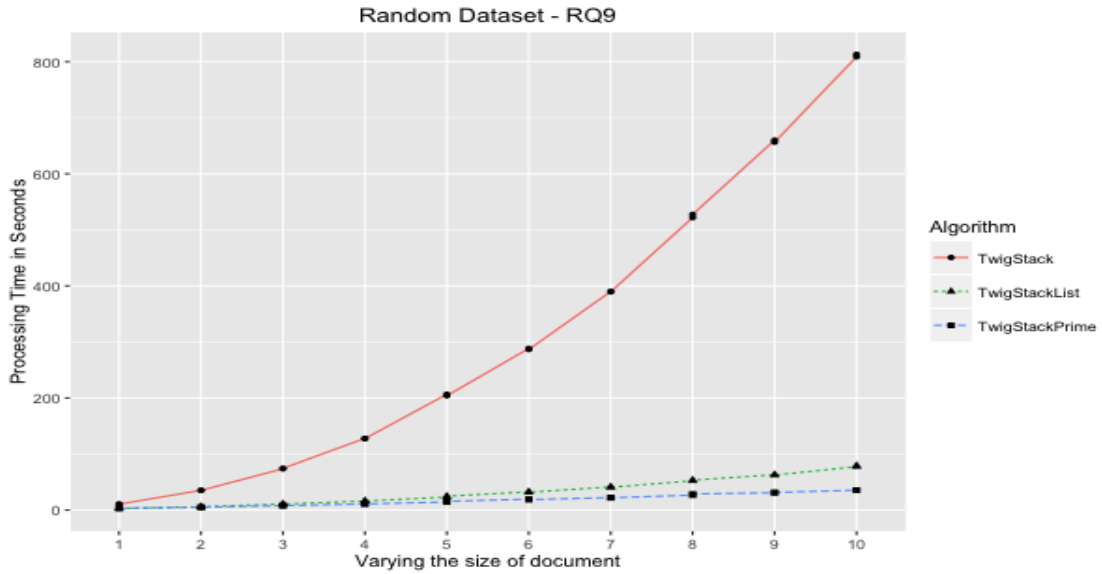


(b)

Figure 6.18: Scalability comparison for XMark datasets.



(a)



(b)

Figure 6.19: Scalability comparison for Random datasets.

algorithms in most cases. The experiment employed twenty nine TPQs over four datasets to evaluate the performance and efficiency of TwigStackPrime against the existing algorithms TwigStack and TwigStackList. According to the experimental results, TwigStackPrime outperformed TwigStack in 18 TPQs and they achieved similar performance in 1 TPQ. When compared with TwigStackList, TwigStackPrime was faster in 19 TPQs and there was no difference in the performance between them in 5 TPQs. Finally, the scalability tests demonstrated that TwigStackPrime can scale well for large datasets. It should be noted that only TwigStack and TwigStackPrime compute answers to TPQs in sorted order as specified in Definition 4.11. In some cases, TwigStackList may produce matches which are not sorted in the root-to-leaf order.

6.6 Conclusion

This chapter presented a new approach that prunes efficiently irrelevant elements for TPQs with P-C relationships. It also introduced a novel holistic algorithm which can utilise the new labelling technique to process TPQs with P-C axes efficiently. Unlike the previous holistic algorithms, TwigStackPrime takes into considerations the CPL relationships between elements in the streams, henceforth it generates fewer path solutions for TPQs with P-C edges. Furthermore, it has been shown analytically that for TPQs with only A-D edges or P-C relationships related to leaf query nodes, TwigStackPrime can guarantee optimal evaluation, in other words, all path solutions contribute to the final result. The overall performance of TwigStackPrime is substantiated in the experiments. In summary, the experimental results have shown that TwigStackPrime has a superior performance to the comparable algorithms in terms of the number of intermediate path solutions and query processing time.

In the chapter that follows, new top-down holistic approaches to process order relationships and sequence operators based on the CPL relationship will be introduced. The next chapter describes the procedures and methods used in this thesis to incorporate the semantics of the order axes into twig patterns.

Chapter 7

Ordered Twig Pattern Matching: Top-Down Approach

7.1 Introduction

This chapter describes the process for developing a new approach to evaluate XPath expressions with ordered axes. The difference between processing unordered and ordered twig patterns was discussed in Chapter 4. In this thesis, the work of [173] will be adapted to extend conventional twig patterns to capture the semantics of order relationships and sequence operators introduced in XPath [222] specification.

The purpose of this chapter is to discuss how ordered twig pattern queries (OTPQs) can be evaluated in a holistic model. It highlights the importance of ordered twig pattern matching. To process OTPQs naïvely, the existing holistic algorithms, such as TwigStack and TwigStackPrime, can be used to compute answers to TPQs without ordered constraints, then merge join operations are performed to guarantee that individual path solutions satisfy ordered predicates of elements. The main weakness with the naïve approach is that it leads to unnecessary computations and a large number of irrelevant path solutions. This chapter provides a systematic way to transfer the path expressions developed, in this thesis (see Section 4.2.2), to order aware twig patterns. A set of algorithms for holistic order twig matching is designed. Finally, experiments are conducted to evaluate the scalability and efficiency of the proposals for ordered twig pattern processing.

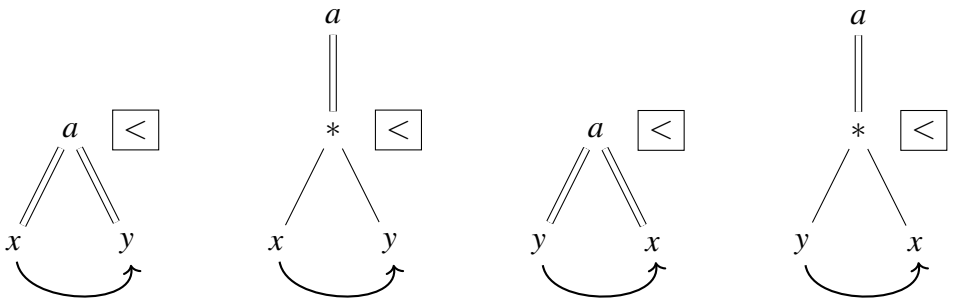
The rest of this chapter is organised as follows: Section 7.2 presents the rewriting rules to transform a standard twig patterns to an ordered twig pattern by incorporating constraints that capture the meaning of order axes and sequence operators. The new algorithms to process ordered TPQs are described in Section 7.3. Section 7.4 is dedicated in the experimental results. The chapter will be concluded in Section 7.5.

7.2 Ordered Twig Pattern

There are four ordered axes, namely *following, following-sibling, preceding* and *preceding-sibling* which can express effectively searches satisfying order constraints among elements of XML trees. These four axes take into consideration both the opening and closing tags of elements because elements with ordered axes must satisfy the document order and must not be nested as was explained in Section 4.2.2. In addition to these ordered axes, sequence operators can be used to express traversals satisfying order which takes into account only the opening tags of elements. In Chapter 4, three constraints were introduced to capture the semantics of ordered axes and sequence operators, namely: Left-to-Right (for short LR) ordering, Sibling Left-to-Right or SLR ordering and Sequence Left-to-Right ordering (abbreviated as SeqLR). In the context of stored data, the existing approaches, such as [188, 141, 145, 118, 189] to process TPQs with ordered axes have been unable to capture the semantics of ordered axes as discussed in Chapter 4.

The representation of XPath expressions with ordered axes as an order aware twig, in this thesis, is based on the ideas presented in [207, 173]. Generally, *preceding* and *preceding-sibling* axes are converted to their equivalent axes *following* and *following-sibling*, respectively. As a result, LR ordering (see Definition 4.19) will be used to encode *following* and *preceding* axes into the equivalent OTPQ, and SLR ordering (see Definition 4.24) utilised to represent the order constraints of *following-sibling* and *preceding-sibling* axes. The SeqLR ordering will be encoded as an array of nodes associated with the context node and used during the query processing. For illustration, consider the following path expression
Q_2 = //a[//y]//f[y « f]. In the graphical notation shown in Figure 7.2c, SeqLR ordering constraint is denoted using a double-headed, dotted arrow from y-node to f-node.

Recall that the Definition 4.28 formalised an ordered twig match addressed in this research study. When an XPath expression contains a step axis with *following-sibling* or *preceding-sibling* axis relating to the context node with a descendant axis, the structures of twig patterns may need to be extended to convey the semantics of these axes. In another way, the relationship between the query node, with *following-sibling* or *preceding-sibling* axis, and its parent query node will be expanded to A-D edges and SLR ordering will be used to represent the order constraints among the query nodes. In contrast, the work of [173] expanded the original query by introducing a new wildcard query node to maintain the sibling relationship among the corresponding sibling query nodes with order relationship. Then the new wild card query node was connected to the ascendant node A-D edge. For example, consider the path expressions
$Q_1 = a//x/following-sibling :: y$ and $Q_2 = a//x/preceding-sibling :: y$, the corresponding order aware twigs to represent these expressions are illustrated in Figure 7.1. The edge between the y-node and its parent a-node is expanded to A-D relationship and an SLR edge is used to preserve the sibling relationship between x-node and y-node in Q_1. By the same token, SLR edge represents the sibling relationship between y-node and



(a) $OTPQ_1$. (b) $OTPQ_1$ [173]. (c) $OTPQ_2$. (d) $OTPQ_2$ [173].

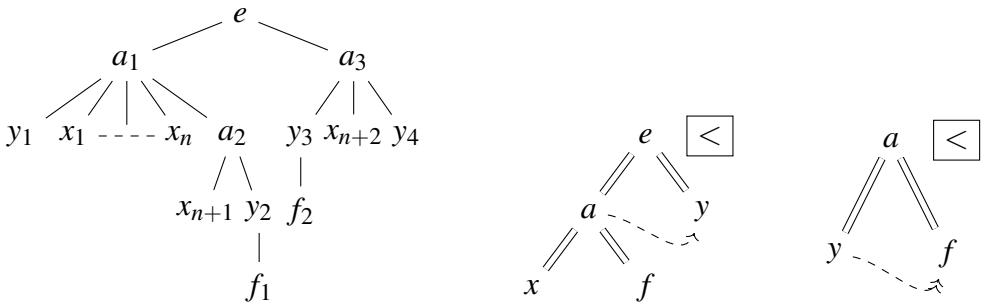
Figure 7.1: Illustration of ordered TPQs with a *following-sibling* or *preceding-sibling* axis following a query node in A-D edge. The solid arrows indicate SLR ordering. (a) and (c) represents the novel approach, whereas b and d shows the work of [173].

x-node in Q_2. For [173], new *-nodes are added to the queries in order to search for sibling elements which are descendants of the ascendant node. In the streaming model, the sibling relationship between two or more elements can not be determined without reading their common parent elements. Consequently, wildcard nodes can not be eliminated. Since this research study aims to improve the query processing in the stored model, the indexing techniques (i.e., labelling scheme) may be used to get rid of wildcard nodes. This novel approach results in widcard-free queries, therefore the number of structural joins are reduced.

As was discussed in Chapter 4, the SLR relationships between elements corresponding to the query nodes with SLR ordering can be computed straight away using the information of parent elements encoded within the labels of elements. That is, the work of [217, 207] exploited the information of parent elements to speed up the process of handling sibling axes in ordered twig queries. Therefore, the labelling scheme proposed in the previous chapter will be extended by incorporating the *parentID* which is the *start* attribute of the parent element in the original labelling scheme. As a result, each element is encoded a 5-tuple as *(start,end,level,parentID,CPL)*. Otherwise, during the parsing process of path expressions, the constraints will be used to record the query nodes which have order relationships and impose sequence operators.

It should be noted that when the root query node in the relative path expression has ordered axes, the root of the XML document (or the document node) will be augmented to the ordered twig as the root query node. This is used to avoid the extensions of the twig to multiple twigs based on sibling axis occurrences. As a result, the cosmic root of the original XML tree can effectively be used to represent an order aware twig in a single twig as has been proved in [173]. The following example illustrates the process for transforming a relative path expression with the root query node imposing ordered axes to an ordered twig pattern.

Example 7.1. *Consider the XML tree in Figure 7.2 and the query* $Q_1 = a[/following :: y][//x]//f$. *During the parsing operation of the path expression, the*



(a) an XML tree T_1. (b) Q_1. (c) Q_2.

Figure 7.2: An example of embedding the document node in the twig to convey the semantics of ordered axes related to the root query node. The dotted arrow indicate the LR ordering constraint. The double-head, dotted arrow indicates the SeqLR constraints for Q_2.

query node a is encountered and recorded as the root. After stripping off the predicate [/following::y], it is found that the relationship between the context-node $\rightarrow$ *a and the current node* $\rightarrow$ *y is order relationship, so the node y is connected to the parent of the context node. Since the context node is the root, the root of the document is added as the root query node to capture the meanings of the ordered axes and assure there is no missing result. The path expression is transformed into* $Q_1 = e//a[/following :: y][//x]//f$. *The twig representation is depicted in Figure 7.2.*

The work in this thesis is motivated by the following observation. The lack of a compact and accurate encoding of path expressions with ordered axes and sequence operators embedded in the conventional twigs hinders algorithms, particularly those for stored data, from effectively processing ordered axes and sequence operators. It is possible that the use of the semantics of ordered and sequence relationships in twig-based algorithms can provide a new solution that makes effective utilization of ordered aware twigs to achieve efficiency. The following subsections describe the notation and data structures used to process ordered axes efficiently. Examples will be also presented to illustrate the difference between order aware TPQ and ordered twig matching in previous approaches.

7.2.1 Notation and Data Structure

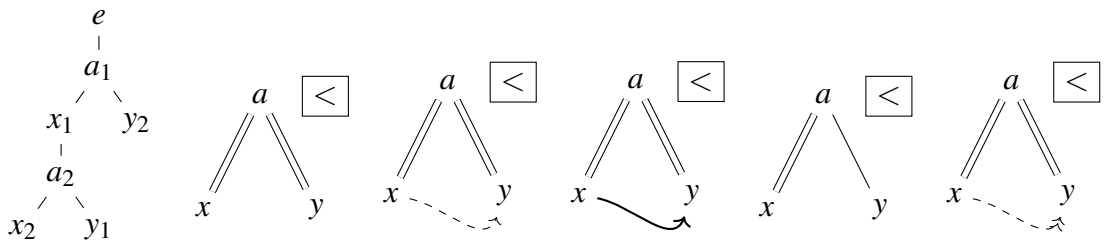
Most of the notation and data structures used in this chapter are the same as those in Section 6.2.1. The only exception is that there are extra, auxiliary functions on nodes of OTPQ to facilitate the ordered twig matching process. Supported functions are as follows: *rightLR(q)* returns all *following* query nodes of q. *rightSLR(q)* returns all *sibling-following* query nodes of q. *rightSeqLR(q)* returns all query nodes which have sequence operators with q. *isOrderedBranching(q)* returns boolean values to see whether q has children with ordered axes or sequence operators or not. *hasOrderingConstraint(q)* returns boolean values to see whether q has ordering constraints or not. In addition to a stack for each query node, the approach proposed in this thesis may make use of an extra data structure, namely lists.

In this chapter, two approaches to process ordered TPQs efficiently are proposed. The first approach extends the original TwigStackPrime to satisfy the SeqLR ordering constraints among TPQs with ordered axes or sequence operators by inspecting only head elements. As a result, the use of stacks in this approach is similar to that in TwigStackPrime explained in Chapter 6. This will be discussed further in Section 7.3.1. On the other hand, the second approach makes use of two types of data structure: stack and list. The basic notion is to buffer elements in main memory to avoid storing irrelevant elements and producing useless path solutions so that the algorithm can check the three ordering constraints. Unlike the first approach, this approach can guarantee, to some extent, most of the elements with ordering constraints match their counterparts. It is likely that reducing the number of irrelevant elements in the streams will provide efficient output enumeration in the second phase. In this approach, each query q which has ordered axes or sequence operators is associated with a list named L_q. The common list operations, such as *empty(), append(), get() and delete()*, are used. At every point during computation, elements in list L_q are sorted in their preorder (i.e., start values).

The next subsection illustrates how the constraints capturing by ordered aware TPQs differ from the constraints maintained by previous ordered twig matching algorithms such as PRIX [188] and OrderedTJ [145], OTJFast in [118], TreeMatch in [146] and TwigPos in [70].

7.2.2 Motivation

This section illustrates how the inability to encode ordered or sequences constraints may results in returning incomplete results by ordered twig matching algorithms. It also demonstrates the limitation of the existing holistic ordered twig matching algorithm proposed in [145] since the other methods adopt their approach in handling order specifications in TPs. Figure 7.3 shows two conventional ordered twigs and two ordered aware twigs.



(a) an XML tree T_2. (b) Q_1. (c) Q'_1. (d) Q_2. (e) Q'_2. (f) Q_3.

Figure 7.3: Ordered Twigs and ordered aware Twigs.

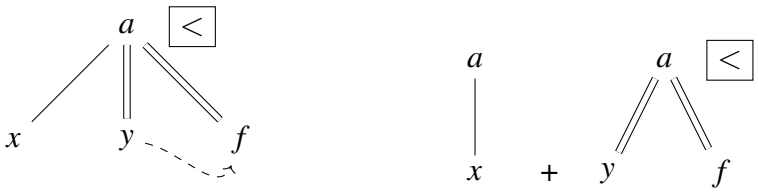
Example 7.2. *The ordered twig in Figure 7.3b represents the following path* Q_1 *=//a//x/following::y. The ordered match yields four structural matches in the XML tree of Figure 7.3a as* (a_1,x_1,y_2),(a_1,x_2,y_1),(a_1,x_2,y_2) *and* (a_2,x_2,y_1). *Figure 7.3c shows an ordered aware twig with LR ordering edge from x-node to y-node which represents the*

same constraint as the one maintained by ordered twig matching, therefore it returns the same matches as the final result. However, the constraints imposed by the two twigs in Figures 7.3d and 7.3f can not be maintained by ordered match of Q_1 because they represent different queries. The twig in Figure 7.3d represents the following query Q_2 = //a//x/following-sibling::y which is looking for x-node which has a sibling y-node, and the x-node must be a descendant of a-node. The conventional ordered twig for this query is presented in Figure 7.3e. The structural matches for Q_2 in the XML tree by maintaining SLR ordering constraint from x-node to y-node yields two matches (a_1,x_1,y_2) and (a_2,x_2,y_1), while the ordered match of Q_2' leads to three matches (a_1,x_1,y_2), (a_1,x_2,y_2) and (a_2,x_2,y_1) which is incorrect since x_2 is not a sibling of the element y_2. The last ordered aware twig depicted in Figure 7.3f is the representation of the following query Q_3 = //a[//x]//y[x«y]. This also can not be maintained by the ordered match of Q_1 since it is looking for x-node and y-node which are descendants of a-node, and x-node must precede y-node in document order only. The structural matches of Q_3 in the XML tree yields five matches as $(a_1,x_1,y_1),(a_1,x_1,y_2),(a_1,x_2,y_1),(a_1,x_2,y_2)$ and (a_2,x_2,y_2).

The above example illustrates that the ordering constraints imposed by ordered aware twigs can not be maintained by ordered match of the standard twigs. The following example demonstrates how ordered aware twigs can represent path expressions with ordered axes or sequence operators in single twigs, instead of extending the conventional twig to a set of twigs.

Example 7.3. *Consider the XML tree in Figure 7.2 and the following query //a[/x]//y/following::f which is looking for a-node that has child relationship with x-node and descendant relationships with y-node and f-node. The y-node must have following relationship with f-node. It can be seen from the semantics of this query, there is no ordered relationship between x-node and its sibling query nodes y-node and f-node. The use of the meanings of ordered axes imposed by the LR ordering constraint leads to a single twig representing the query as in Figure 7.4a. The straightforward approach to process this query using the existing holistic ordered matching algorithms would represent the query in two twigs. The unordered twig is to represent the simple path //a/x. On the other hand, the ordered twig is to capture the ordered constraint between y-node and f-node in the sub-query //a//y/following::f. To find the structural matches of the query, the results of the unordered and ordered twigs are merged by identifying the common prefixes to compute answers to the query. The main disadvantages of this approach is that representing the query using two twigs can lead to a proportional increase in time and space requirement of query evaluation. In contrast, the ordered aware twig in Figure 7.4a has n matches as $(a_1,x_1,y_1,f_1),\ldots$ and (a_1,x_n,y_1,f_1).*

Before proceeding to introduce the new approaches, it is important to illustrate the limitation of the previous holistic ordered twig matching algorithm, OrderedTJ [145] which is an extension to TwigStackList to process ordered queries. The authors introduced a



(a) Q_4. (b) Q'_4.

Figure 7.4: Ordered aware Twig in (a) and conventional twig representation of Q_4 in (b).

new concept, called ordered children extension (for short OCE) to control the number of intermediate path solutions in ordered TPQs. That is an element likely involves in ordered queries if it has OCE as defined in Definition 7.4. The OCE is an extension of children extension of TwigStackList discussed in Chapter 3.

Definition 7.4 (Ordered children Extension)**.** *A query node q has the ordered children extension if the following properties hold:*

- $\forall\, n_i \in childrenAD(q)$, *there is an element* e_i *which is the head of* T_{n_i} *which refers to the tag streaming of query node* n_i*, and a descendant of* e_q*, and* e_i *also has OCE.*
- $\forall\, n_i \in childrenPC(q)$, *there is an element* e_i*, which is the head of* T_{n_i}*, has a parent in the path from* e_q *to* e_i *in the corresponding buffering list* L_q *of query node q, and* e_i *also has OCE.*
- $\forall\, n_i, \ldots, n_{i_{k-1}} \in children(q)$ *which are ordered (*n_i *to* n_k*),* e_{n_i} *satisfies the following relationship with* $e_{n_{i+1}}$ *that is the end value of the head element* e_{n_i} *is less than the start value of the head element* $e_{n_{i+1}}$*.* e_{n_i} *also has OCE.*

In comparison with the the original TwigStackList, the only extension is to check the ordered constraints imposed by the following relationship among elements corresponding to branches in ordered TPQs. The next example shows why using the following relationship as the ordered constraint in OrderedTJ [145] gives problems regardless of whether A-D or P-C edges are used.

Example 7.5. *Consider the XML tree* T_2 *of Figure 7.3a, and the following ordered queries* Q_1 *= //a//x/following::y and* Q_2 *= //a/x//following::y. If following relationship is embedded within the filtering phase (i.e., getNext() with OCE), errors may occur. Initially, the head elements are* $a \rightarrow a_1$*,* $x \rightarrow x_1$ *and* $y \rightarrow y_1$*. For* Q_1*,* a_1 *satisfies the first two properties since there is no P-C edge in the query as the buffering list is empty. The ordered constraint imposed between its child nodes is found to be violated by* x_1 *because it does not end before* y_1*, hence,* x_1 *is discarded. This leads to three matches of* Q_1 *in the XML tree* T_2 *as* (a_1, x_2, y_1)*,* (a_1, x_2, y_2) *and* (a_2, x_2, y_1)*. In contrast, the query has four matches in the XML tree* T_1 *as the OrderedTJ algorithm missed the match* (a_1, x_1, y_2)*. For* Q_2*,* a_1 *is found to satisfy the OCE as it has descendant relationship with* x_1 *and the element* y_1 *has a parent* a_2 *in the path from* a_1 *to* y_1*. However, the child elements of* a_1 *do not satisfy the following relationship which leads to skip the element* x_1*. This results in returning only one match of* Q_2 *in* T_2 *as* (a_2, x_2, y_1) *and losing the match -* (a_1, x_1, y_2)*.*

In the following section, the thesis proposes a new concept to filter out irrelevant elements form the streams which unlikely contribute in the final results for ordered aware TPQs. The new approaches can overcome the limitation in the previous approaches and extends TwigStackPrime, introduced in Chapter 6, to process ordered queries efficiently. To avoid the generation of useless path solutions and storing irrelevant elements, some elements may be buffered to the main memory using simple lists.

7.3 Holistic Ordered Twig Matching algorithms

In this section, the thesis introduces new approaches which permit a combination of different techniques to match ordered TPQs as addressed in Definition 4.28. First it extends the child and descendant extension used by TwigStackPrime to process unordered TPQs for the purpose of considering the ordered relationships among head elements imposed by the generalised case. In ordered aware twigs, there are three additional constraints to capture the semantics of ordered axes and sequence operators, namely LR, SLR and SeqLR ordering constraints defined in Definitions 4.19, 4.24 and 4.26, respectively. It may be observed that the SeqLR relationship is the generalised relationship for the other constraints. That is, an advanced preorder filtering algorithm should only take into account the document order among elements with these constraints to avoid missing potential results. In this thesis, a simple extension to TwigStackPrime is proposed to handle ordered constraints by examining head elements only. This new approach is called OTJPrime, Ordered Twig Join Prime, which can skip efficiently elements violating SeqLR relationships. The other one is OTJPrimeList which can be seen as an extension to the first one. It uses lists to buffer some elements with ordering constraints to avoid storing irrelevant elements. Unlike OTJPrime, OTJPrimeList performs strict filtering for the ordering constraints as specified by the ordered queries.

The next subsections describe the novel approaches and introduce a new approach to incorporate the data structure proposed in [207], which enables efficiently the checking of *following-sibling* relationships in the queries, into the holistic approaches.

7.3.1 Ordered Twig Matching Algorithm: OTJPrime

This section presents a novel top-down holistic algorithm to process OTPQs. The OTJPrime algorithm extends TwigStackPrime to consider the minimal ordered constraint (i.e., the SeqLR ordering constraint which is the less restrictive relationship among the others) which does not affect the query answers. The new algorithm can be seen as alternative to TwigStackPrime when the queries containing ordered axes and sequence operators.The majority of the OTJPrime algorithm is the same as the TwigStackPrime. OTJPrime differs in the use of the SeqLR ordering constraint inside the *getNext()* function and extending the second phase to merge paths on their common prefixes as well as check all the ordering constraints to compute answers to OTPQs.

As was discussed in Chapter 6, *getNext* is a core function as the advanced preorder filtering strategy in twig pattern matching. It is used to identify the next element associated with the query node to be processed and advanced in its corresponding stream. A key to the practical performance of *getNext* in TwigStackPrime is that elements returned must have a child descendant extension according to Definition 6.17 which can prune efficiently a considerable number of irrelevant elements. As the basic axes can be checked using the child and descendant extension, the ordered extension, defined in Definition 7.6, aims to take into consideration the SeqLR ordering constraint among children of ordered, branching elements to check whether or not they likely contribute to the final results.

Definition 7.6 (Ordered Extension)**.** *A query node q has the ordered extension if the following properties hold:*

- $\forall\ n_i \in rightLR(q)$*, there is an element e_i which is the head of T_{n_i} and has a start value greater than the start value of e_q which is the head of T_q.*
- $\forall\ n_i \in rightSLR(q)$*, there is an element e_i which is the head of T_{n_i} and has a start value greater than the start value of e_q which is the head of T_q.*
- $\forall\ n_i \in rightSeqLR(q)$*, there is an element e_i which is the head of T_{n_i} and has a start value greater than the start value of e_q which is the head of T_q.*

Algorithm 7 shows the general framework for identifying potential elements participating in the final results of OTPQs extending *getNext(q)* proposed in Chapter 6, see Algorithm 5. In OTJPrime, the branching elements must have their children with ordering constraints sorted in ascending order of their start values in addition to satisfy A-D and P-C relationships to be consider for further processing. As a result, *getNext(q)* returns an element e_q of a query node $q \in OTPQ$ with four properties. The first three properties are inherited from the child and descendant extension while the fourth property aims to check the ordered extension introduced in this chapter.

i e_q has a descendant element e_{q_i} in each of the streams corresponding to its child elements where e_{q_i} is the head element of a query node q_i = *children(q)* (this property is checked in Lines 7-9).

ii each of its child elements satisfies recursively the first property (this property is checked in Lines 2-4).

iii if q has Parent-Child edge(s) with its child query nodes, then e_q has a child e_{q_i} in T_{q_i} for each query node q_{q_i} = childrenPC(q) (this property is checked in Lines 21-23 of *getElement* function, see Algorithm 5).

iv if q has child nodes with ordering constraints, then each of its child elements with the ordering constraints has ordered extension according to Definition 7.6. this property is checked in Lines 13-21).

The description of the new extension to the original *getNext()* is as follows: at Line 13, it checks if the current query node has child nodes with the ordering constraints. if so, it iterates over its children to assure that they all have the ordered extension. Note that by getting to Line 13, the current query node q has already satisfied the three properties of the child and descendant extension. According to Lemma 6.20, all the head elements corresponding to the child query nodes of q are parts of solution involving q. Since elements are sorted in their start values, the ordered relationship between two elements is violated when the following node occurs before its preceding node. Thus, it is useless and can be discarded safely. At Lines 17, 19 and 21, if a query node pointed by LR, SLR or SeqLR edges precedes the source of the edge, it is returned to the main algorithm in which advances the corresponding stream to point to the next element and proceeds to the next cycle.

Algorithm 7: getNext(q)

```
   Input: q is a query node
   Result: a query node in OTPQ which may or may not be q
1  if isLeaf(q) then
   |  return : q
2  foreach node n_i in children(q) do
3  |  g_i = getNext(n_i)
4  |  if g_i ≠ n_i then
   |  |  return : g_i
5  n_max = a query node with the maximum start value ∈ children(q)
6  n_min = a query node with the minimum start value ∈ children(q)
7  while getEnd(getElement(q)) < getStart(getElement(n_max)) do
8  |  advance(q)
9  if getStart(getElement(q)) > getStart(getElement(n_min)) then
   |  return : n_min
10 else
11 |  // the following code lines add the ordered extension for child nodes with the
   |    ordering constraints
12 |  // if an element is found to violate the order of children with ordering constraints,
   |    the algorithm returns it to the main algorithm and it can be discarded safely.
13 |  if isOrderedBranching(q) then
14 |  |  foreach node n_i in children(q) do
15 |  |  |  if hasOrderingConstraint(n_i) then
16 |  |  |  |  foreach node m_j in rightLR(n_i) do
17 |  |  |  |  |  if getStart(getElement(n_i)) > getStart(getElement(m_j)) then
   |  |  |  |  |  |  return : m_j
18 |  |  |  |  foreach node m_j in rightSLR(n_i) do
19 |  |  |  |  |  if getStart(getElement(n_i)) > getStart(getElement(m_j)) then
   |  |  |  |  |  |  return : m_j
20 |  |  |  |  foreach node m_j in rightSeqLR(n_i) do
21 |  |  |  |  |  if getStart(getElement(n_i)) > getStart(getElement(m_j)) then
   |  |  |  |  |  |  return : m_j
   |  return : q
```

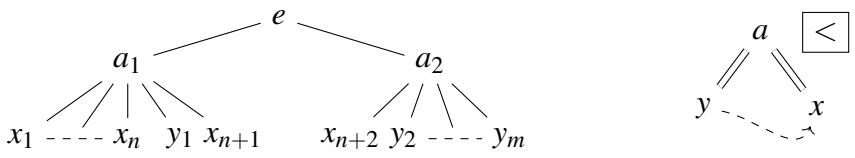
It should be noted that in the second phase, path solutions in the output arrays are merge-joined based on their common branching query nodes and child elements must satisfy the ordering relationships as specified by the ordered queries. As a result, query matches are returned as the query result. Note that OTJPrime can process efficiently both unordered and ordered queries because Lines 13-22 in the *getNext* function can be skipped if the OTPQ does not have ordered branching query nodes.

In comparison with TwigStackPrime, which does not take into account the ordered constraints between child query nodes, the effect of OTJPrime can be illustrated in the following example.

Example 7.7. *Consider the XML tree in Figure 7.5a, and the ordered query Q_1 = // a //y/following::x which is represented in Figure 7.5b. The head elements in their streams are $C_a \rightarrow a_1$, $C_y \rightarrow y_1$ and $C_x \rightarrow x_1$. For TwigStackPrime, The first call of getNext() inside the main algorithm will return $a \rightarrow a_1$ because it has child and descendant extension. The cursor of the query node is shifted to the next element a_2, therefore all the elements which are parts of solution involving a_1 will be pushed to their corresponding stacks to generate individual root-to-leaf paths. For the branching element a_1, the algorithm will produce 1 path solution for the simple path //a//y and n+1 path solutions for the path //a//x. In the same way, a_2 has a child and descendant extension, hence it is returned to the main algorithm. When all elements have been processed, TwigStackPrime will generate m-1 paths for the simple path //a//y rooted at the element a_2, and 1 single path for the non-predicate path //a//x. In the second phase, the algorithm merges the paths based on their common ancestor and the single paths ending with the y-node must satisfy the LR relationship with the corresponding elements in the root-to-leaf paths ending with the x-node. According to the LR ordering constraint, the following paths $(a_1,x_1),\ldots,(a_1,x_n)$ violate the following relationship with (a_1,y_1) since it is the only path for their common ancestor a_1. Thus, they can not contribute to the final results. For a_2, it has one path ending with the x-node which occurs before all the paths ending with the y-node, hence all its paths can not be merge-joined. However, Q_1 has only one match in the tree as (a_1,y_1,x_{n+1}). Unlike TwigStackPrime, the first call of getNext() inside the main algorithm, OTJPrime will not return the elements $x_1,\ldots,x_n$ because they violate the SeqLR relationship with respect to the element y_1 as the head element of the y-node. After n calls of getNext(), the head elements are $C_a \rightarrow a_1$, $C_y \rightarrow y_1$ and $C_x \rightarrow x_{n+1}$. The element a_1 is found to have the child and descendant extension and its child query nodes satisfy the ordered extension. Therefore, the algorithm pushes a_1 into its corresponding stack and advances the cursor to a_2. Then, both y_1 and x_{n+1} are pushed into their stacks to generate the individual root-to-leaf paths. Now, the cursors point to the elements $C_a \rightarrow a_2$, $C_y \rightarrow y_2$ and $C_x \rightarrow x_{n+2}$. a_2 has the child and descendant extension but x_{n+2} violates the SeqLR relationship with y_2, therefore it is returned by getNext() to be skipped and advance the cursor of x-node to the next element. Once the cursor of x-node is advanced and points to the empty label $\perp$, OTJPrime terminates the first phase. In the second phase, there are only two*

path solutions which are merge-joined because they satisfy the LR (i.e., the following) relationship. It can be observed that the effect of OTJPrime to avoid processing the subtree rooted at a_2 and generating a large number of useless paths was efficient. To sum up, the straightforward approach using TwigStackPrime produced ((n+2) + m) paths, while the OTJPrime algorithm generates only two paths which contribute to the final result.

The above example illustrates the effect of the OTJPrime algorithm which considers the simplest level of ordered relationship among sibling query nodes with ordering constraints. This novel approach can avoid processing irrelevant elements and the generation of useless paths. However, this approach has a serious drawback. That is, if the head elements satisfy the SeqLR relationship but they do not strictly satisfy the ordering relationships as specified by the ordered queries. Hence, the algorithm may be inefficient. Example 7.8 illustrates this limitation.



(a) an XML tree T_3. (b) Q_1.

Figure 7.5: Illustration to ordered extension.

Example 7.8. *Consider the XML tree in Figure 7.6, and the ordered query in Figure 7.5b. Assume the head elements are $C_a \rightarrow a_2$, $C_y \rightarrow y_2$ and $C_x \rightarrow x_{n+2}$. The algorithm returns x_{n+2} In order to skip over it because it occurs before y_2 and violates the SeqLR ordering relationship. In the next cycle, the head elements are $C_a \rightarrow a_2$, $C_y \rightarrow y_2$ and $C_x \rightarrow x_{n+3}$. The algorithm returns a_2 because it has the child and descendant extension and the head element y_2 precedes x_{n+3} in document order. Thus, elements which are parts of solution involving a_2 will be pushed to their stacks. Although there is no match of the query in the subtree rooted at a_2, the algorithm fails to avoid the storage of useless paths. This case demonstrates how the head elements can be ordered but they do not contribute to the final result.*

It can be observed from Examples 7.5 and 7.8 that if the algorithm restricts the relationship among the head elements, some useful results may be missed as the OrderedTJ algorithm fails to give sufficient consideration to the hierarchical representation of data that XML provides [145]. On the other hand, applying a weak ordering restriction by using the SeqLR relationship may generate a number of useless paths. Accordingly, the next section introduces a new approach which performs a strict filtering using simple buffering techniques partially inspired by [207, 144].

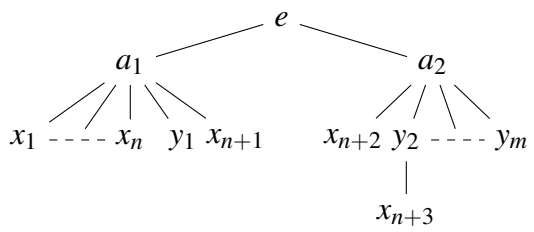


Figure 7.6: An example of inefficient processing using the ordered extension among head elements.

7.3.2 Ordered Twig Matching Algorithm: OTJPrimeList

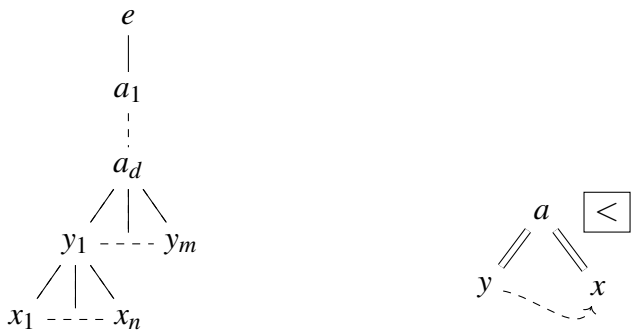
In this section, OTJPrimeList is proposed to overcome the limitations of the OTJPrime algorithm where many useless paths may be generated. OTJPrimeList exploits the buffering technique proposed in TwigStackList [144] to store elements with ordering constraints in order to perform a strict filtering as required by ordered queries. It also can be considered as a combination of TwigStackList and OTJPrime to improve the efficiency of ordered twig matching. The OTJPrimeList algorithm introduces two improvements to the previous TwigStackList and OTJPrime algorithms:

1. irrelevant elements are pruned from their temporary streams if they do not satisfy the ordering constraints as specified by the ordered queries.

2. elements are returned in a sorted order conforming to the document order.

These improvements are implemented by means of lists so that the algorithm makes multiple scans over elements with ordering constraints if and only if their common ancestor has the child and descendant extension and the ordered child query nodes have the ordered extension. The number of scans is bounded by the maximum number of ordering constraints acquired by sibling query nodes. In addition, OTJPrimeList makes an extension to the *getElement* function by considering the first element of the list associated with the ordered query node as the head element if the list is not empty. Otherwise, the element pointed to by the cursor is treated as the head element as in OTJPrime. The following example demonstrates how the novel approach can filter out irrelevant elements from the streams and in turn improve the efficiency.

Example 7.9. *Consider the XML tree T_4 and the ordered query Q_1 in Figure 7.7. In the first d cycles of OTJPrime, all elements a_1 to a_d are pushed into the stack S_a because they have child and descendant extensions and the element y_1 has the ordered extension as $y_1 \prec x_1$. Therefore, The OTJPrime algorithm produces $d \times (n+m)$ path solutions even though there is no match of the ordered query in the XML tree. On the contrary, OTJPrimeList firstly assures that the parent query node must satisfy the child and descendant extension and if it has a child query node with ordering constraints, it has to have the ordered extension. Hence, the element a_1 satisfies the four properties so that all elements in the T_y and T_x streams which are descendants of a_1 are buffered to their lists for further investigation. Before proceeding to the next iteration by getNext, the algorithm visits all*

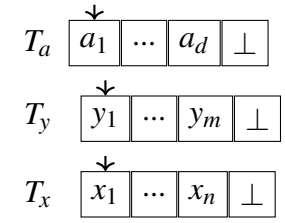


(a) an XML tree T_4 (b) Q_1

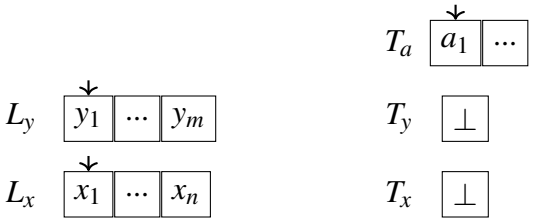
Figure 7.7: Hard case with restricted memory (i.e., the OTJPrime algorithm). It can not be known whether $x_1,\ldots,x_n$ are useless before y_1 is processed, or whether $y_1,\ldots,y_m$ are useless before $x_1,\ldots,x_n$ are processed.

elements in the L_y and L_x to eliminate useless elements. Since elements are found not to satisfy the ordered relationship as specified by the query, they are removed from the lists. Consequently, OTJPrimeList terminates the evaluation before proceeding to the next cycle because streams of leaf query nodes pointed at the end as cursors are pointed to the position after the last element in the streams (i.e., no more elements to be processed). OTJPrimeList does not produce intermediate results for this query. Figure 7.8 presents instances during the execution of OTJPrimeList for Q_1 on the XML tree T_4. Note that a_1 is pushed into the stack to avoid the extra filtering process for its descendants with the same tag, namely $a_2,\ldots,a_d$.

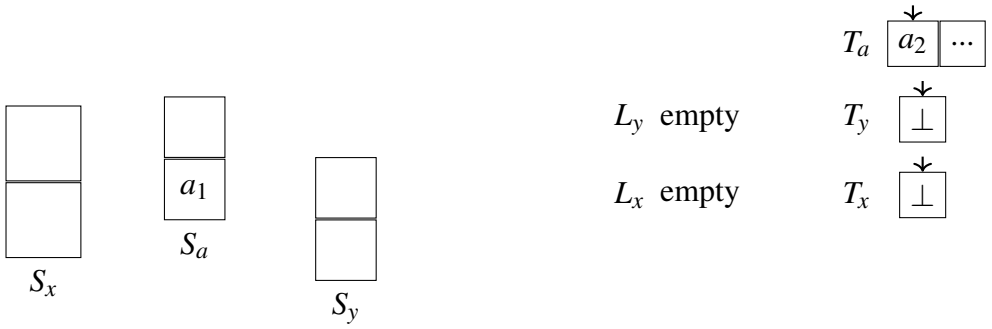
The above example provides an overview of how the OTJPrimeList algorithm improves the filtering strategy of OTJPrime by eliminating useless elements which either violate the ordering relationships or are not parts of a solution involving their preceding/following (*preceding-sibling/following-sibling*, respectively) elements. It can be observed that elements in the buffering lists might be visited multiple number of times in order to avoid the generation of useless paths. As a result, the run time complexity for the filtering phase becomes $O(|P| \times |F| + |Input|)$ where P is the sum of lengths of input lists for query nodes with ordering constraints and F is the sum of lengths of input lists for query nodes pointed by ordering constraints, while Input is the sum of the lengths of the remaining input lists without ordering constraints. However, in most cases, this approach can guarantee linear enumeration of the outputs and increases the overall performance by avoiding unnecessary merge joins which are computationally expensive. The effect of this novel approach is shown in the experimental evaluation presented in Section 7.4. It should be noted that the algorithm considers only a small part of the streams to be buffered and this can be performed only once for subtrees of the original XML tree. In addition, the buffering lists are split for query nodes connected by the SLR edge constraints or below P-C edges with the SeqLR constraints to enable access usable element matches in ordered TPQs with the *following-sibling* and *preceding-sibling* axes. Therefore, there is one list for each elements' level in the data tree. Each element can identify the corresponding list by retrieving the



(a) First call of *getNext* inside OTJPrimeList.



(b) Buffering elements with ordering constraints by extraFiltering(a) function.

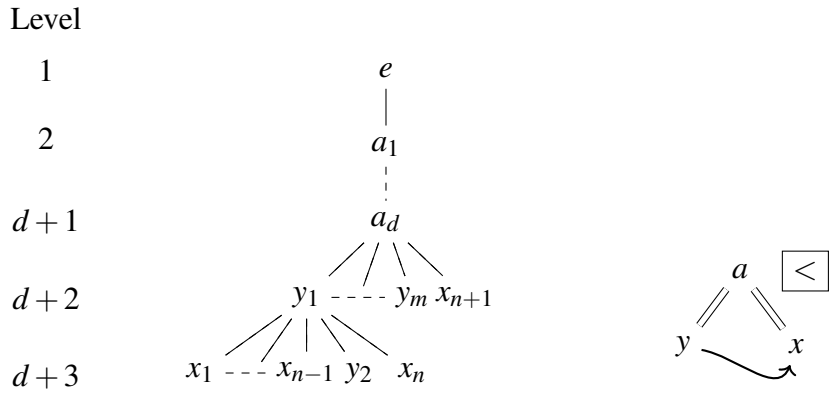


(c) OTJPrimeList processing of Q_1 over T_4 of Figure 7.7.

Figure 7.8: Illustration to the extraFiltering function used by OTJPrimeList. (a) and (b) illustrate the use of temporary streams for query nodes y and x. (c) depicts the status after filtering the streams.

list of its depth. The new version of the OTJPrimeList is called, OTJPrimeMultiLists which indicates the utilisation of multi-lists to buffer elements for each query node which has or is pointed to by the SLR constraints as well as it is below P-C axis and has the SeqLR constraint. Note that the filtering strategy used by OTJPrimeMultiLists algorithm is comparable to that used by OTJPrimeList but it is expected to allow efficient handling of sibling axis. Example 7.10 demonstrates the effect of OTJPrimeMultiLists to avoid scanning irrelevant elements in the buffering lists.

Example 7.10. *Consider the XML tree T_5 and the ordered query Q_2 = //a//y/following-sibling::x in Figure 7.9. After OTJPrimeMultiLists ensures that element a_1 satisfies the child and descendant extension as well as $y_1 \prec x_1$, it appends all elements which are descendants of a_1 to their level lists because y-node and x-node are restricted with the SLR constraint. As illustrated in Figure 7.10, the extra filtering strategy scans elements in the level split lists of y-node to see whether they satisfy the SLR relationships with at least one element in the corresponding list of x-node using the level as a key. To check whether or not y_1 contributes to the final result, the algorithm needs to visit only one element stored in the list of level d+2, namely x_{n+1}. In contrast, OTJPrimeList has to visit at least n elements before finding that y_1 is useful with x_{n+1}. In similar fashion, y_2 matches with x_n so that $x_1, \ldots, x_{n-1}$ are removed from the list because they do not participate in any solution. When the algorithm filters out all irrelevant elements which*



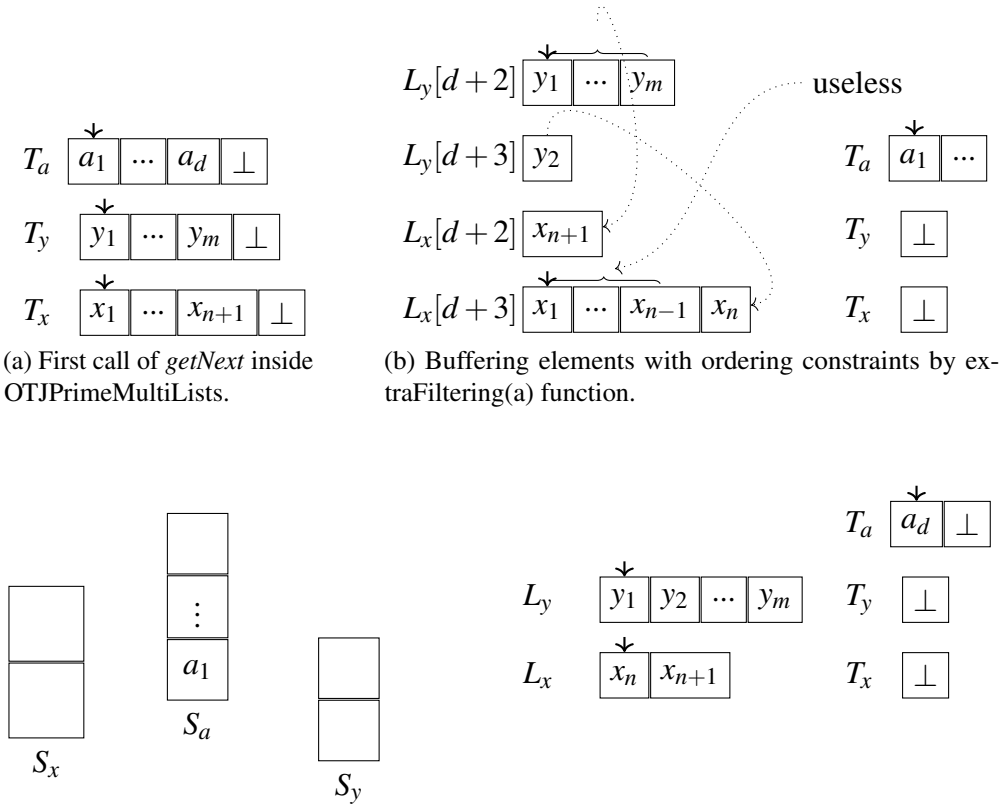
(a) an XML tree T_5 (b) Q_2

Figure 7.9: An example to illustrate the effect of OTJPrimeMultiLists.

are parts of solution involving a_1, the multi lists are combined in a single list containing elements sorted by their start values. Both OTJPrimeList and OTJPrimeMultiLists produce only path solutions which are merge-joined, therefore the merge join operation is linear with respect to the number of paths.

Algorithm 8 extends the *getNext* function used by OTJPrime to filter out elements which do not contribute to the final result through extra filtering pass at Line 23 as the first twenty two lines are similar to that in Algorithm 7. It is implemented by cleaning buffering lists from left to right and in-place overwriting of elements not satisfying the ordering constraint specified by the ordered queries. The three ordering constraints among elements corresponding to sibling query nodes are checked thoroughly in the *extraFiltering* function shown in Algorithm 9. Firstly, it appends only elements which are descendants of the current element of query node q by calling the *moveToList* function (see Algorithm 10) at the first line. The algorithm identifies two cases where elements are appended to level split lists. The first case is the existence of a SLR constraint between query nodes so that elements corresponding to these query nodes are stored in level split lists. The second case is that two sibling query nodes are connected to their parent query node with P-C axes and they have the SeqLR constraint. For the second case, the algorithm checks the SeqLR relationship between sibling elements using their *parentID* attributes to ensure that they have a common parent. The *getElement* function is also extended to maintain the head element returned depending on whether the list is empty or not. If the list is empty, the head element is the one pointed to by the cursor. Otherwise, the head element is the first element in the list stored at index 0. Likewise, elements in the list are shifted by removing the first element while in the streams, which do not have buffering constraints or have an empty list, are advanced by moving the cursor forward.

The subsection below shows the correctness of the algorithms proposed. It also analyses their complexities.



(a) First call of *getNext* inside OTJPrimeMultiLists.

(b) Buffering elements with ordering constraints by extraFiltering(a) function.

(c) OTJPrimeMultiLists processing of Q_2 over T_5 of Figure 7.9.

Figure 7.10: Illustration to extraFiltering function used by OTJPrimeMultiLists.

7.3.3 Analysis of Ordered Twig Matching Algorithms

This section presents the correctness of the new algorithms and analyses their complexities. The correctness of OTJPrime, OTJPrimeList and OTJPrimeMultiLists follows from the correctness of TwigStackPrime described in Section 6.4.2. The *getNext* functions used by these algorithms are extensions to that of TwigStackPrime which takes into account only A-D and P-C relationships. The new *getNext* functions assure that the head elements of query nodes with ordering constraints must have start values less than their following elements.

Definition 7.11 (Head element)**.** *In OTJPrime, for each query node q in a TPQ Q, the element indicated by the cursor C_q is the head element of q.*

Definition 7.12 (Head element)**.** *In OTJPrimeList and OTJPrimeMultiLists, for each query node q in an ordered TPQ Q, if the list L_q is not empty, then the first element of L_q is the head element of q. Otherwise, the element indicated by the cursor C_q is the head element of q.*

Definition 7.13 (Ordered Child and Descendant Extension)**.** *A query node q has the ordered child and descendant extension if the following properties hold:*

- $\forall\, n_i \in childrenAD(q)$, *there is an element e_i which is the head of T_{n_i} and a descendant of e_q which is the head of T_q.*
- $\forall\, n_i \in childrenPC(q)$, *there is an element e_q which is the head of T_q and its CPL parameter is divisible by tagPrime(n_i).*
- $\forall\, n_i \in children(q)$, *$n_i$ must have the ordered child and descendant extension.*
- $\forall\, n_i \in children(q) \wedge hasOrderingConstraints(n_i)$, *$n_i$ must have the ordered extension according to Definition 7.6.*

The above definitions are essential to establish the correctness of the following lemmas:

Lemma 7.14. *Suppose $getNext(q)$ returns a query node q' which is pointed by one of the ordering constraints and $q \neq q'$ at either Lines 18, 20 or 22 of getNext. Then, there is no further solution involving some elements of children(parent(q')) which have start values greater than the start value of the head element of q'.*

Proof. Suppose that on contrary, there is a new solution using some elements in the streams of children(parent(q')) which have start values larger than the start value of q'. Using range-based property, it is known by the SeqLR ordering relationship that the start value of the following query node must be greater than the start value of the preceding query node, therefore all elements in the streams of children(q') must have start values larger than the start value of the head element of q', which is a contradiction. □

(a) an XML tree T_6 (b) Q_3

Figure 7.11: An example to illustrate Lemma 7.14.

Example 7.15. *Consider the XML tree T_6 and the ordered query Q_3 = //a//x//preceding::y in figure 7.11. The first call of getNext ensures that a_1 has the ordered child and descendant extension so that it is returned and pushed into the stack S_a. After that, y_1 and x_1 are pushed into their stacks to produce the path solutions. Then, x_2 is returned because it violates the ordered child and descendant extension of its parent a_2. Henceforth, the next iteration skips a_2 because it does not have an element of x-node. After that, y_2 is returned. Finally, OTJPrime produces 3 useful path solutions and 1 useless path. Q_3 has two matches in T_6 as (a_1, y_1, x_1) and (a_1, y_1, x_2).*

Lemma 7.16. *For any arbitrary query node q' which is returned by getNext(q), the following properties hold:*

1. *q' has the ordered child and descendant extension.*
2. *Either (a) $q = q'$ or (b) q' violates the ordered child and descendant extension of the head element e_q of its parent(q').*

Proof. (Induction on the number of ordered child and descendants of q). If q is a leaf query node, it is returned in Line 2 because it verifies all the properties 1 and 2 in Lemma 7.16. Otherwise, the algorithm recursively gets g_i = *getNext(n_i)* for each child of q in Line 4. If for some i, there is $g_i \neq n_i$, it is known by the inductive hypothesis g_i verifies the properties 1 and 2b with respect to q, so the algorithm returns g_i in Line 6. Otherwise, by inductive hypothesis that all q's child nodes satisfy properties 1 and 2a with their corresponding sub-queries. At *getElement(q)* (Lines 21-25), *getNext* advances from T_q all segments that do not satisfy the divisibility by the product of prime numbers in $childrenPC(q)$ returned from *getQCPL*. After that, the algorithm advances from T_q (Lines 9-10) all segments that are beyond the maximum start value of $n_i \in children(q)$. Then, if q does not satisfy properties 1 and 2a, Line 9 guarantees that $n_i \in children(q)$ with the smallest start value satisfies properties 1 and 2b with respect to start value of q's head element e_q is returned. Otherwise, q is checked to see whether or not it has child query nodes with ordering constraints at Line 13. If q is found to be an ordered, branching query node, the algorithm compares start values for each child of q with ordering constraints against their following elements at Lines 14-22. If for some j, there is $m_j \prec n_i$, and it is known by inductive hypothesis that m_j verifies the properties 1 and 2b with respect to q, hence the algorithm returns m_j in Lines 18, 20 and 22. Otherwise, it is known by inductive hypothesis that

all q's child nodes satisfy properties 1 and 2a with their corresponding sub-queries and following elements. After that q is returned. □

The above lemmas guarantee that each time *getNext* returns a query node q and the head element of q does not have the ancestor extension, the current head element of q can be skipped and the cursor of q can be advanced to the next element in the stream of q. Using the above lemmas and lemmas introduced in Section 6.4.2, the next theorem will be used to prove the correctness of OTJPrime, OTJPrimeList and OTJPrimeMultiLists and their core functions *getNext*.

Theorem 7.17. *Given an ordered twig pattern query Q and an XML document D, Algorithms OTJPrime, OTJPrimeList and OTJPrimeMultiLists correctly return answer to Q on D.*

Proof. In Algorithms OTJPrime, OTJPrimeList and OTJPrimeMultiLists, *getNext(root)* is repeatedly invoked to determine the next query node to be processed. Using Lemma 7.16, it is known that all elements returned by $q_{act} = getNext(root)$ have either the ordered child and descendant extension or violated the ordered child and descendant extensions of their parents. If $q_{act} \neq root$, Line 4, the algorithm pops from $S_{parent(q_{act})}$ all elements that are not ancestors of the head element of q_{act} by Lemma 6.21. In addition, it is already known q_{act} has an ordered child and descendant extension so that Line 5 checks whether $S_{parent(q_{act})}$ is empty or not. If so, it indicates that it does not have the ancestor extension, and it can be discarded safely to continue with the next iteration. Otherwise, the current head element of q_{act} has both the ancestor and ordered child and descendant extensions which guarantee its participation in at least one root-to-leaf path. Then, $S_{q_{act}}$ is cleaned by popping elements which do not contain the head of q_{act}, using Lemma 6.22. Then, the item in the stack is used to maintain pointers from itself to the query root. Finally, if q_{act} is a leaf query node, all possible combinations of single paths with respect to q_{act} can be computed at Lines 8-9 and stored in the corresponding output array. To ensure that answers returned are correct with respect to the ordered axes and sequence operators, the postprocessing step to merge path solutions must check the ordering constraints as specified by the ordered query to avoid generating incorrect matches. □

The correctness holds for TPQs with ordering constraints in addition to both Ancestor-Descendant and Parent-Child relationships. In contrast to top-down unordered twig matching algorithms, the second phase must merge individual path solutions based on their common prefixes and elements corresponding to query nodes with ordering constraints that match their following elements as specified by the ordered query. For example, to evaluate the ordered query Q_4 on the XML tree T_7 shown in Figure 7.12. The merge operation must prevent the path (a_1, y_2) to be merged with the path (a_1, x_1) even though the two paths are useful and contribute to the final result.

Moving on now to consider the run time complexity of the proposal algorithms, it should be noted that none of the algorithms can guarantee, for a specific class of query,

(a) an XML tree T_7 (b) Q_4

Figure 7.12: An example of ordered query Q_4. The path solution ending at y_1 can merge with the path solutions ending at x_1 and x_2, while the path solution involving y_2 can only be merged with x_2 to compute answers to Q_4 in T_7.

linear evaluation with respect to the size of input and output lists since they consider ordering branching query nodes as the key to filter out irrelevant elements. That is, elements with ordered axes or sequence operators may be gathered in the input streams after buffering them to the lists and satisfy the LR or SeqLR ordering relationships with elements in subtrees rather than their tight subtrees. That is, the buffering lists may contain elements with ordered axes or sequence operators which are found to satisfy the LR or SeqLR ordering relationships with elements in subtrees rooted from ancestors which are not their *lowest common ancestor* within a deeply recursive XML tree. The OTJPrimeList and OTJPrimeMultiLists can ensure in some cases that all path solutions produced are merge-joined but this can not be considered as optimal processing but the worst case behaviour for the algorithms is $O(|P| \times |F| + |Input|)$ where P is the sum of lengths of input lists for query nodes with ordering constraints and F is the sum of lengths of input lists for query nodes pointed to by ordering constraints, while Input is the sum of the lengths of the remaining input lists without ordering constraints. On the other hand, the OTJPrime algorithm can process ordered queries less efficiently than the buffering algorithms with linear performance expected in the filtering phase (i.e., the first phase). Moreover, the purpose of the OTJPrimeList's and OTJPrimeMultiLists's extra filtering technique are to speed up the query processing. It is likely that this does not bring an overhead in comparison to the existing approaches to process ordered TPQs.

Similar to TwigStackPrime, the worst case space complexity of the OTJPrime algorithm is proportional to the longest path in the XML tree times the number of query nodes in the ordered TPQ. By contrast, the worst case space complexity of the OTJPrimeList and OTJPrimeMultiLists algorithms are equal to the longest path in the XML tree times the number of query nodes in the ordered TPQ plus the sum of lengths of input lists corresponding to query nodes with the ordering constraints.

The next section describes the experiments to test the performance of the top-down ordered holistic twig join algorithms just described.

Algorithm 8: getNext(q)

Input: q is a query node
Result: a query node in TPQ which may or may not be q

1 // Starting from Line 22 in Algorithm 7
2 **if** *isOrderedBranching(q)* $\wedge$ *(empty(S_q)* $\vee \neg$ *ancestor(S_q.get(0),getElement(q)))* **then**
3 | extraFiltering(q) // see Algorithm 9
return : q
4 **Function** `getQCPL(`*Query node q*`):`
5 | // the prime number assigned to the query node which is the product of its child query node prime numbers
6 | qCPL = 1
7 | **foreach** *node n_i in childrenPC(q)* **do**
8 | | qCPL = qCPL $\times$ tagPrime(n_i)
| **return :** qCPL
9 **Function** `checkCursor(`*Query node q*`):`
10 | **if** *childrenPC(q) > 0* **then**
11 | | **while** $\neg$ *eof(C_q)* $\wedge$ *getCPL(C_q) % getQCPL(q)* $\neq$ *0* **do**
12 | | | $C_q \rightarrow C_q + 1$
13 | **if** *eof(C_q)* **then**
| | **return :** $\infty, \infty, \infty, \infty, 1$ // out of range label
14 | **else**
| | **return :** C_q // the element pointed by C_q
15 **Procedure** `advance(`*Query node q*`):`
16 | **if** $\neg$ *empty(L_q)* **then**
17 | | L_q.delete(0) // remove the first element
18 | **else**
19 | | $C_q \rightarrow C_q + 1$
20 **Procedure** `allElements(`*Query node q*`):`
21 | returns a special iterator to all the elements for a query node pointed by the SLR ordering edge sorted in their start values.
22 **Function** `getElement(`*Query node q*`):`
23 | **if** *(isRoot(q)* $\vee$ *q* $\in$ *childrenAD(parent(q)))* **then**
24 | | **if** $\neg$ *empty(L_q)* **then**
| | | **return :** L_q.get(0)
25 | | **else**
| | | **return :** checkCursor(q)
26 | **else**
27 | | **if** $\neg$ *empty(allElements(q))* **then**
| | | **return :** allElements(q).get(0)
28 | | **else**
| | | **return :** checkCursor(q)

Algorithm 9: extraFiltering(q)

Input: q is a query node
Result: perform a strict filtering as specified by the ordered TPQ the ordering constraints among potential elements

```
1  moveToList(q) // see Algorithm 10
2  foreach node n_i in children(q) do
3      if hasOrderingConstraint(n_i) then
4          foreach node m_i in rightLR(n_i) do
5              i = j = 0
6              while i < L_{n_i}.size do
7                  if L_{n_i}[i] has at least one element satisfying the LR relationship in L_{m_i}
                     then
8                      L_{n_i}[j] = L_{n_i}[i]
9                      j = j + 1
10                 i = i + 1
11             resize(L_{n_i},j)
12             // Scan L_{m_i} to prune elements which are not parts of ordered extensions
                 for elements in L_{n_i}
13         // the only difference between OTJPrimeList and its refined version
             OTJPrimeMultiLists is in the following constraints as the level can be used
             for the current element
14         foreach node m_i in rightSLR(n_i) do
15             i = j = 0
16             while i < L_{n_i}.size or allElements(n_i).size do
17                 level = getLevel(L_{n_i}[i])
18                 if L_{n_i}[i] has at least one element satisfying the SLR relationship in
                     L_{m_i} or L_{m_i}[level] then
19                     L_{n_i}[j] = L_{n_i}[i]
20                     j = j + 1
21                 i = i + 1
22             resize(L_{n_i},j)
23             // Scan L_{m_i} or L_{m_i}[level] to prune elements which are not parts of ordered
                 extensions for elements in L_{n_i}
24         foreach node m_i in rightSeqLR(n_i) do
25             i = j = 0 while i < L_{n_i}.size or allElements(n_i).size do
26                 level = getLevel(L_{n_i}[i])
27                 if L_{n_i}[i] has at least one element satisfying the SeqLR relationship in
                     L_{m_i} or L_{m_i}[level] then
28                     L_{n_i}[j] = L_{n_i}[i]
29                     j = j + 1
30                 i = i + 1
31             resize(L_{n_i},j)
32             // Scan L_{m_i} or L_{m_i}[level] to prune elements which are not parts of ordered
                 extensions for elements in L_{n_i}
```

Algorithm 10: moveToList(q)

Input: q is a query node

Result: fetch some elements from the streams which are dscendants of the current element corresponding to the query node q

```
1  foreach node n_i in children(q) do
2      if hasOrderingConstraint(n_i) then
3          // append all elements in the stream T_{n_i} to L_{n_i} if they are descendants of the
             current element in the stream T_q or the first element in L_q if q has ordering
             constraints.
4          while ¬ eof(n_i) ∧ ancestor(getElement(q),C_{n_i}) do
5              level = getLevel(C_{n_i})
6              // if the current child n_i has or is pointed by SLR edge constraint so that it
                 is appended to level split data structure, otherwise to the regular list.
7              if isPointedBySLR(n_i) ∨ (n_i ∈ childrenPC(q) ∧ isPointedBySeqLR(n_i) ∧
                 n_i ∈ rightSeqLR(m_i) ∧ m_i ∈ childrenPC(q)) then
8                  L_{n_i}[level].append(C_{n_i})
9              else
10                 L_{n_i}.append(C_{n_i})
11             C_{n_i} → C_{n_i} + 1
```

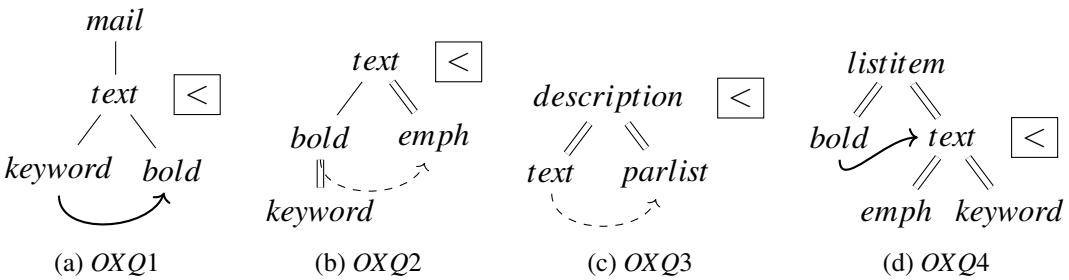
7.4 Experimental Evaluation

The following experiments explore the effect of the ordered child and descendant extension, extra filtering strategy and multi list data structure to process ordered TPQs. Since the proposed algorithms are the first top-down holistic approaches which can prune useless elements from the streams prior to the merge join operation, the unordered top-down holistic algorithms used in the experiments of Chapter 6 are modified to use the straightforward postprocessing approach in their second phases. Hence, this section provides the experimental results of the performance comparison of top-down twig join algorithms, namely SFTwigStack, SFTwigStackList, SFTwigStackPrime, OTJPrime, OTJPrimeList and OTJPrimeMultiLists. The first three algorithms use the straightforward post-processing which is abbreviated as the prefix SF. In contrast, OTJPrime utilises the ordered child and descendant extension in the filtering phase to minimise the number of useless paths, while OTJPrimeList and OTJPrimeMultiLists extends OTJPrime by applying an extra filtering strategy to provide efficient merge join in the second phase. Similar to the experiments in Chapter 6, all the algorithms tested in the experiments were implemented and added to the query processor described in Section 5.2.2.2.

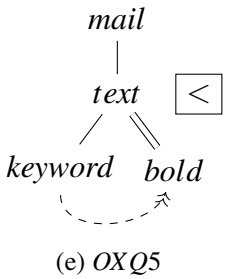
XMark, TreeBank and Random datasets were used for the experiments. The benchmarked datasets used in the experiments and their characteristics are shown in Section 6.5.1.1 of Chapter 6. The ordered TPQs on XMark and TreeBank datasets were obtained from [145, 118], and then modified to carry the meanings of ordered axes and sequence operators. On the other hand, the ordered TPQs over the Random dataset were created by adding the ordered axes and sequence operators to the previously unordered TPQs used in Section 6.5.1.1. For the sake of simplicity, ordered TPQs were encoded so that the code indicates the dataset and its ordered TPQs. By way of illustration, OXQ1 refers to the first ordered TPQ issued over XMark dataset. The characteristics of the ordered queries over XMark are presented in Figure 7.13. The properties of TPQs selected over TreeBank are given in Figure 7.16. Figure 7.19 provides an overview of the ordered TPQs over Random dataset. The experiments compare the algorithms based on metrics similar to those described in Section 6.5.1.2. Text-based queries are provided in Appendix B.

7.4.1 Experimental Results

The section highlights the evaluation of the experimental results. The first step in this process was to inspect all the results returned from the algorithms. Since the algorithms produced the same results, the validity of the new approaches can be verified. By way of illustration, the discussion of the query performance related to a particular dataset is presented within an individual subsection. The query performances for OTPQs over XMark, TreeBank and Random datasets are evaluated in Sections 7.4.1.1, 7.4.1.2 and 7.4.1.3, respectively. The scalability tests are provided in Section 7.4.1.4.



(a) $OXQ1$ (b) $OXQ2$ (c) $OXQ3$ (d) $OXQ4$



(e) $OXQ5$

Code	Useful Paths	Result size
OXQ_1	8611	6581
OXQ_2	3806	2516
OXQ_3	15821	11955
OXQ_4	0	0
OXQ_5	16381	16034

(f) Features of OXQs

Figure 7.13: Ordered TPQs tested over XMark dataset. The useful paths are the root-to-leaf paths which can be merged in order to produce the final result. "Result size" is the matching result of a twig pattern query.

7.4.1.1 XMark

The number of intermediate path solutions when running the XMark dataset are given in Figure 7.14. An immediate observation from the figure is that the new approaches are more efficient in terms of the intermediate results than the straightforward approaches for all ordered queries on this dataset. OTJPrimeList and OTJPrimeMultiLists showed a superior performance in avoiding the generation of useless paths whereas OTJPrime failed in some cases to produce only useful path solutions. However, its performance, in general, is significantly superior to the straightforward approaches. To assess the query performance, the Kruskal-Wallis test was carried out to see whether or not there is a performance difference between two algorithms at least for every ordered TPQ on the dataset. The results of the groups analysis are set out in Table 7.1. It can be seen from the data in Table 7.1 that a significant difference between two groups (i.e., algorithms) at least was evident in all queries.

Table 7.1: Results for the comparison groups on the XMark dataset.

Query	p-value	p-value < 0.05
OXQ1	4.62E-119	TRUE
OXQ2	1.90E-118	TRUE
OXQ3	1.17E-122	TRUE
OXQ4	1.99E-123	TRUE
OXQ5	2.52E-111	TRUE

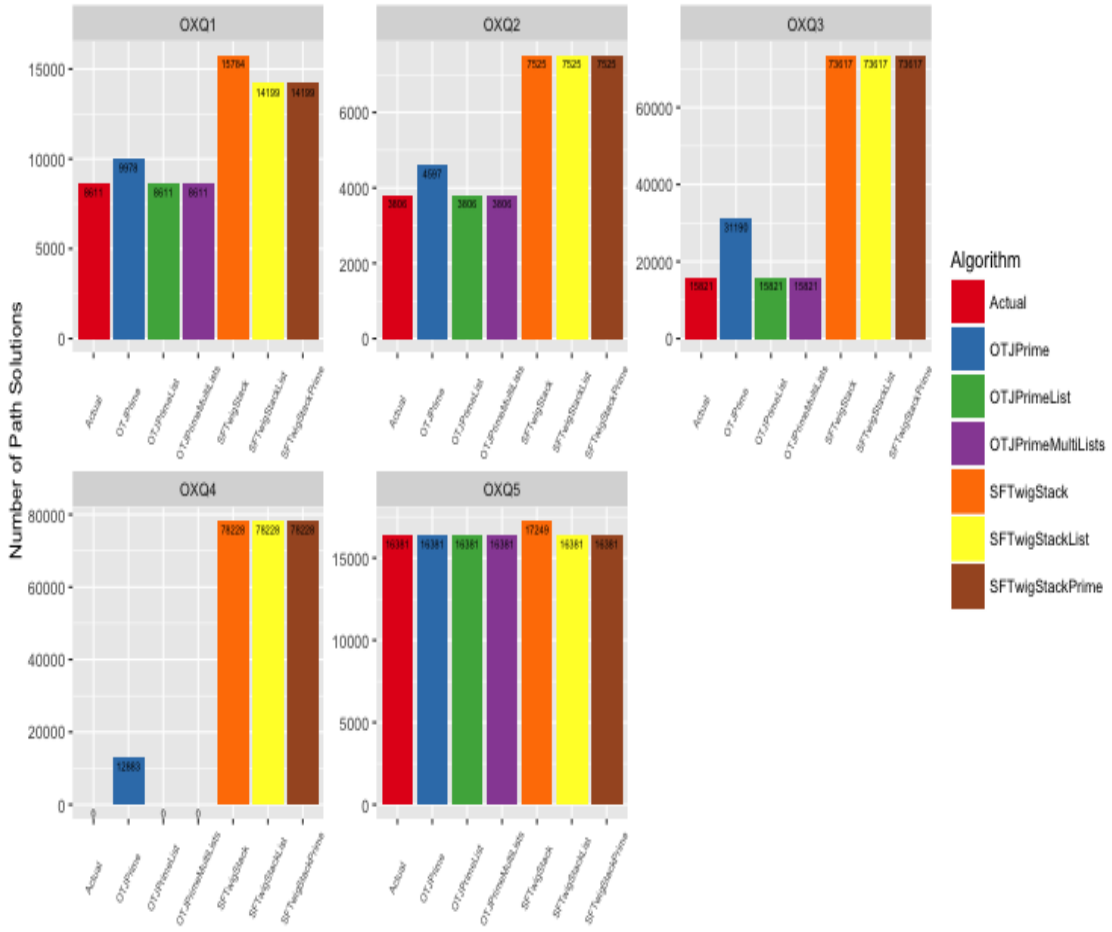


Figure 7.14: The number of intermediate path solutions generated by each algorithm for the queries tested over XMark. "Actual" represents the number of paths appearing in the final matches.

Consequently, the number of paired comparisons for this dataset can be obtained using Formula 6.2 described in Chapter 6 as $= \frac{(6\times(6-1))}{2} \times 5 \ = 75$. The full results of the pairwise comparisons can be found in Appendix B. As shown in Figure 7.15, the new algorithms were significantly faster than the other straightforward approaches on all OTPQs. According to the results provided in Table 6.12, OTJPrimeList had the best performance in all cases followed by its refined version OTJPrimeMultiLists. Even though OTJPrime applied a weak filtering among ordered, sibling query nodes, it had a superior performance to the straightforward approaches and ran slightly slower than OTJPrimeList and OTJPrimeMultiLists. Note that OTJPrime has an advantage over the buffering techniques that it does not consume an extra amount of main memory in the filtering phase. When OTJPrime compared with the straightforward approaches, it ran in a range from two to thirty times faster than the algorithms based on the merge operation to match ordering constraints. The buffering algorithms, in turn, were at least twice fast as OTJPrime. For each pairwise comparison, the effect size suggested that there is a medium to large practical significance unless the null hypothesis is rejected, it suggested that there is low practical significance.

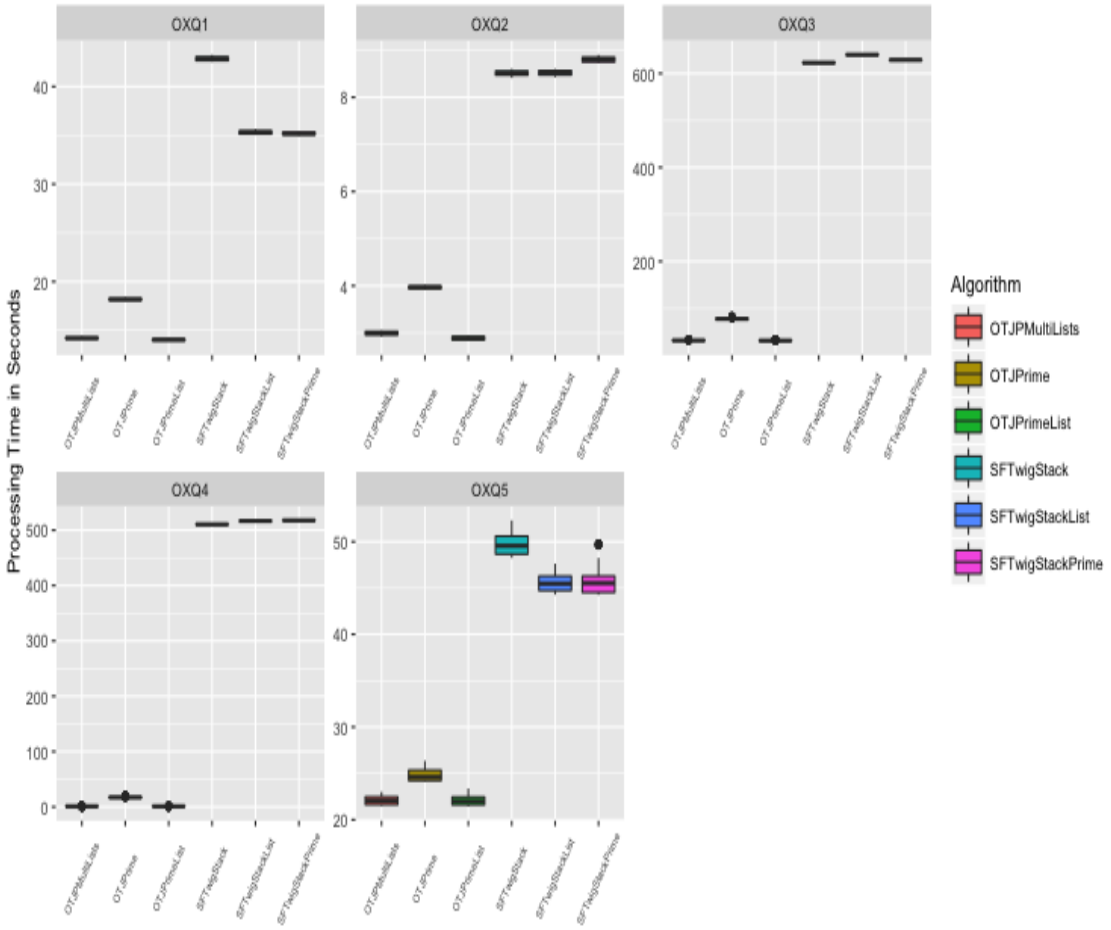


Figure 7.15: Query processing time of the algorithms compared for OTPQs against XMark.

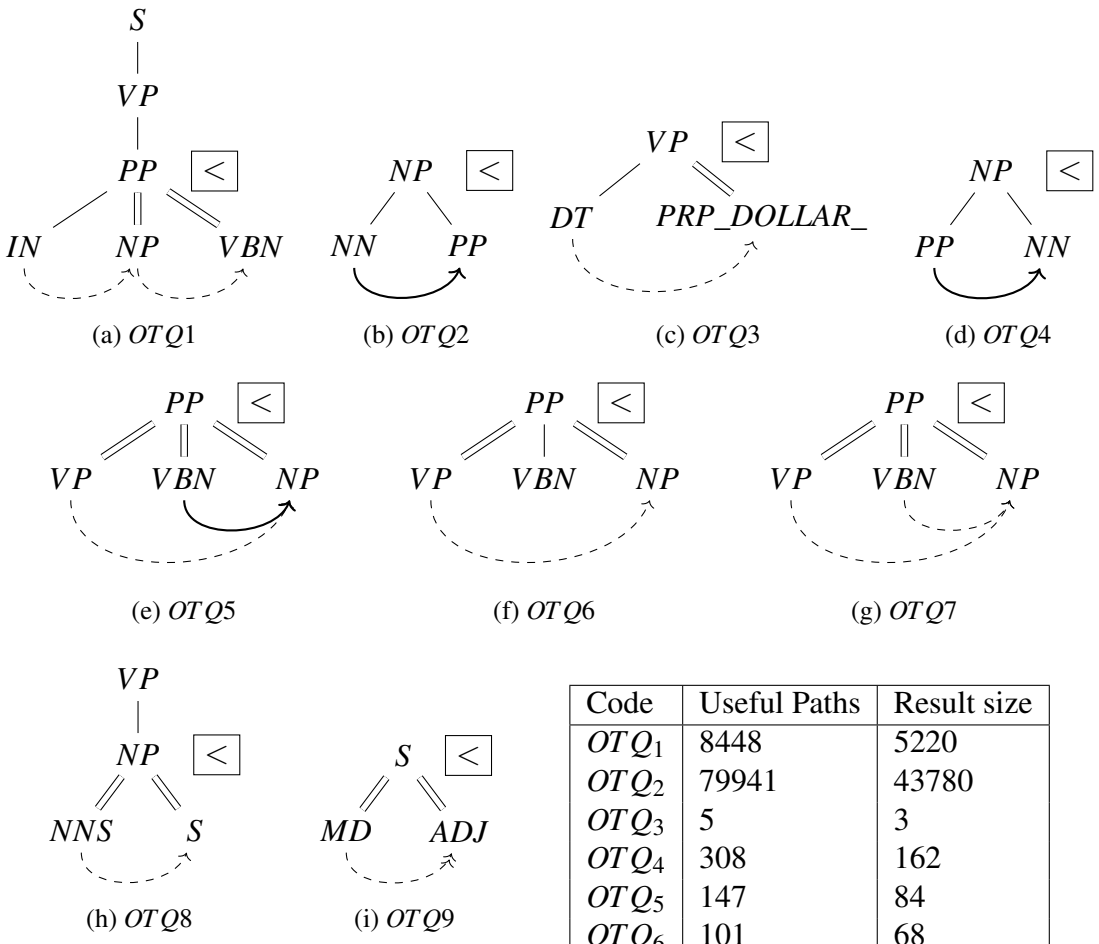
Table 7.2: The overall comparisons based on U tests over XMark dataset. "-" indicates no statistically difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
SFTwigStack	5	19	1
SFTwigStackList	4	19	2
SFTwigStackPrime	4	20	1
OTJPrime	15	10	0
OTJPrimeList	24	0	1
OTJPMultiLists	20	4	1

To conclude, the experimental results demonstrated that the novel approaches to match ordered TPQs had a superior performance to the other techniques using postprocessing in terms of the number of paths generated and query running time. For the XMark dataset, the extra filtering strategy applied by OTJPrimeList and OTJPrimeMultiLists did not bring any overhead and they were multiple times faster than the comparable algorithms. For OTPQs with SLR ordering constraints in OXQ_1 and OXQ_4, OTJPrimeMultiLists failed to improve the efficiency of sibling matching when compared with OTJPrimeList. The most obvious finding to emerge from the analysis is that OTJPrime can be considered to evaluate ordered

TPQs on dataset of this type when the main memory is limited so that only the elements in a single path can be loaded. To summarize, OTJPrimeList significantly outperformed in all queries the other algorithms with the exception of OXQ_3 where it had similar performance to OTJPrimeMultiLists as can be seen in Table 7.2.

7.4.1.2 TreeBank



(a) $OTQ1$ (b) $OTQ2$ (c) $OTQ3$ (d) $OTQ4$

(e) $OTQ5$ (f) $OTQ6$ (g) $OTQ7$

(h) $OTQ8$ (i) $OTQ9$

Code	Useful Paths	Result size
OTQ_1	8448	5220
OTQ_2	79941	43780
OTQ_3	5	3
OTQ_4	308	162
OTQ_5	147	84
OTQ_6	101	68
OTQ_7	6674	6271
OTQ_8	12784	9188
OTQ_9	35	19

(j) Features of OTQs

Figure 7.16: Ordered TPQs tested over TreeBank dataset. The useful paths are the root-to-leaf paths which can be merged in order to produce the final result. "Result size" is the matching result of a twig pattern query.

For the TreeBank dataset, the experiment is designed to test ordered TPQs which have different structures and combinations of A-D, P-C, LR, SLR and SeqLR edges. These queries were generated specifically to give a comprehensive comparison of algorithms. The number of intermediate results produced by each algorithm is depicted in Figure 7.17. From the illustrative graph in Figure 7.17, it can be seen that by far the best performance is achieved by the buffering algorithms. However, they produced useless paths in ordered

TPQs with LR ordering constraints, namely OTQ_1, OTQ_5, OTQ_7 and OTQ_8. These results are consistent with the theoretical analysis discussed in Section 7.3.3. Despite the fact that the TreeBank dataset has many recursive tags, OTJPrime surprisingly was found to perform efficiently in most cases by eliminating a considerable number of useless single paths. It seems possible that these results are due to the fact that elements in the TreeBank dataset are ordered in the sequence they occur in the original text so that the SeqLR ordering constraint can prune irrelevant elements efficiently.

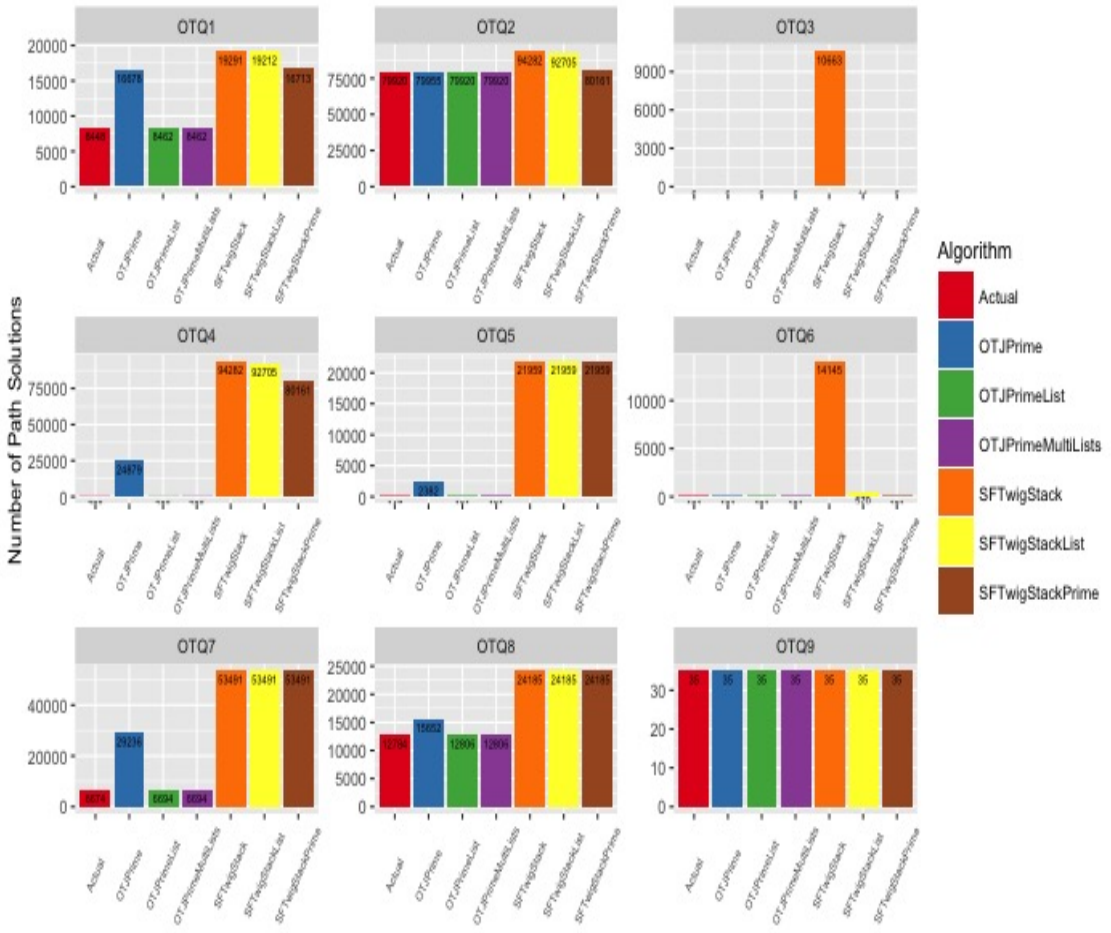


Figure 7.17: The number of intermediate path solutions generated by each algorithm for the queries tested on the TreeBank dataset. "Actual" represents the number of paths appearing in the final matches.

To evaluate the query performance, the Kruskal-Wallis tests presented in Table 7.3 revealed that there is a difference in the performance between at least two algorithms. Accordingly, paired comparisons based on the U test of Mann Whitney were computed. The number of pairwise comparisons for this dataset can be obtained using Formula 6.2 as $= \frac{(6\times(6-1))}{2} \times 9 = 135$. The full results of the pairwise comparisons can be found in Appendix B. The effect of the extra filtering can be seen in OTQ_4 in which OTJPrimeList and OTJPrimeMultiLists consumed less than two seconds while the straightforward approaches needed more than one thousand seconds. However, the straightforward approaches are comparable to the new algorithms in queries which examine relatively ordered subtrees

Table 7.3: Results for the comparison groups on TreeBank dataset.

Query	p-value	p-value < 0.05
OTQ1	8.56E-114	TRUE
OTQ2	3.33E-127	TRUE
OTQ3	3.93E-121	TRUE
OTQ4	2.21E-125	TRUE
OTQ5	6.17E-124	TRUE
OTQ6	3.01E-118	TRUE
OTQ7	1.03E-115	TRUE
OTQ8	5.18E-120	TRUE
OTQ9	4.98E-19	TRUE

within the original tree such as OTQ_1, OTQ_3, OTQ_6 and OTQ_9. The useful number of path solutions is highlighted in Table 7.16j.

Table 7.4: The overall comparisons based on U tests over TreeBank dataset. "-" indicates no statistically difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
SFTwigStack	13	30	2
SFTwigStackList	15	28	2
SFTwigStackPrime	19	25	1
OTJPrime	28	16	1
OTJPrimeList	33	10	2
OTJPMultiLists	22	21	2

To sum up, OTJPrime and OTJPrimeList showed a superior performance to the other algorithms in terms of the number of intermediate results and query running time for deeply recursive dataset. From the data in the illustrative graph in Figure 6.15 and Table 7.4, it was observed that OTJPrime and OTJPrimeList were faster than their counterparts in 28 and 33 cases out of 45 for each algorithm, respectively.

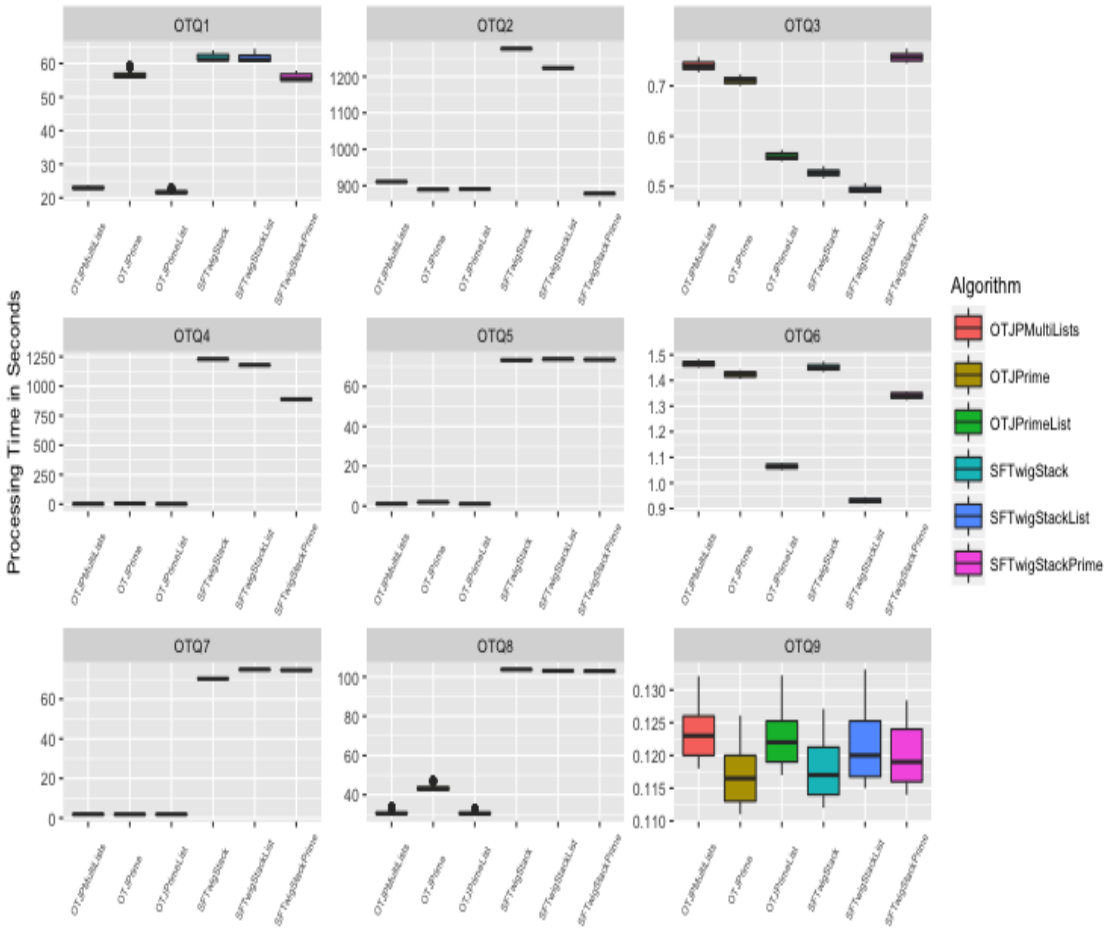
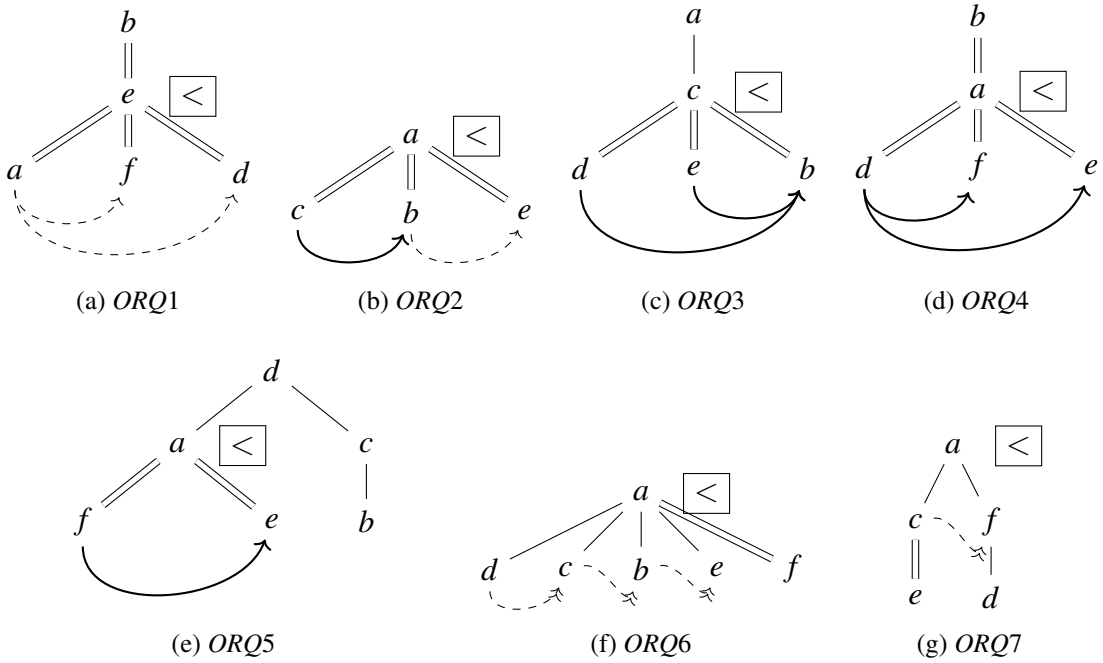


Figure 7.18: Query processing time of the algorithms compared for OTPQs on TreeBank.



(a) *ORQ*1 (b) *ORQ*2 (c) *ORQ*3 (d) *ORQ*4

(e) *ORQ*5 (f) *ORQ*6 (g) *ORQ*7

Code	Useful paths	Result size
ORQ_1	2482	2069
ORQ_2	3271	1994
ORQ_3	3007	1196
ORQ_4	4398	2562
ORQ_5	806	403
ORQ_6	8786	3757
ORQ_7	2457	1906

(h) Features of ORQs

Figure 7.19: Ordered TPQs tested over Random dataset. The useful paths are the root-to-leaf paths which can be merged in order to produce the final result. "Result size" is the matching result of a twig pattern query.

7.4.1.3 Random

In the Random dataset, the experiment is to evaluate the performance of the algorithms on a deeply recursive XML tree but less complicated than TreeBank. The number of intermediate results are shown in Figure 7.20. Similar to the experimental results obtained from the previous datasets, the buffering approaches produced approximately the same path solutions as the useful paths. In addition, the OTJPrime algorithm showed a superior performance in avoiding useless paths despite the complexity of this dataset in terms of structure. In general, the effect of the new approaches is significantly superior to the straightforward approaches. Figure 7.21 depicts query processing overall performance for this experiment. To evaluate the query performance, the Kruskal-Wallis test was carried out to test the null hypothesis stating that there is no difference in the performance between the compared algorithms. The results of the groups analysis are presented in Table 7.5.

From the data in Table 7.5, every Kruskal-Wallis test revealed that there is a significant difference between two algorithms at least.

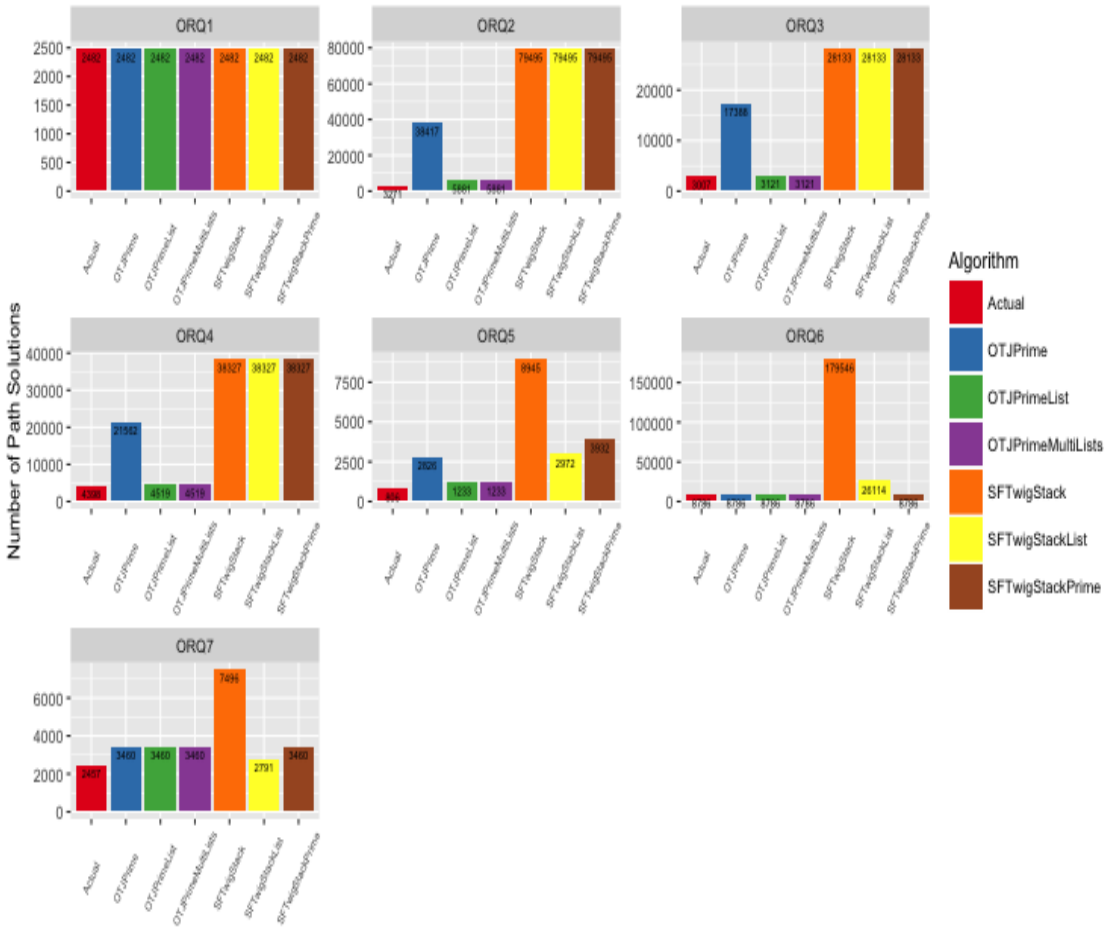


Figure 7.20: The number of intermediate path solutions generated by each algorithm for the queries tested on the Random dataset. "Actual" represents the number of paths appearing in the final matches.

Table 7.5: Results for the comparison groups on the Random dataset.

Query	p-value	p-value < 0.05
ORQ1	2.80E-115	TRUE
ORQ2	2.88E-125	TRUE
ORQ3	1.98E-121	TRUE
ORQ4	3.42E-122	TRUE
ORQ5	1.16E-115	TRUE
ORQ6	3.33E-127	TRUE
ORQ7	3.55E-118	TRUE

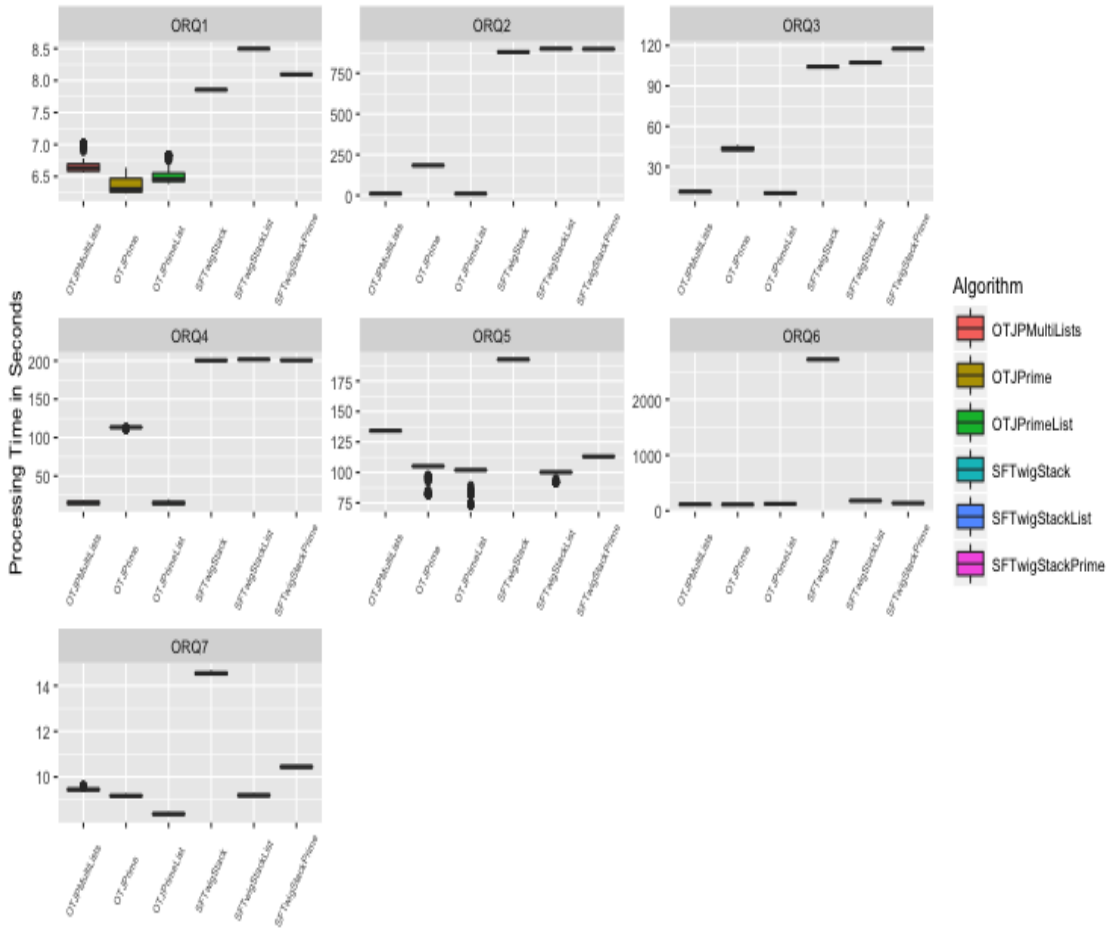


Figure 7.21: Query processing time of the algorithms compared for OTPQs on TreeBank.

Table 7.6: The overall comparisons based on U tests over the Random dataset. "-" indicates no statistically difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
SFTwigStack	8	27	0
SFTwigStackList	10	25	0
SFTwigStackPrime	8	27	0
OTJPrime	26	9	0
OTJPrimeList	30	4	1
OTJPMultiLists	22	12	1

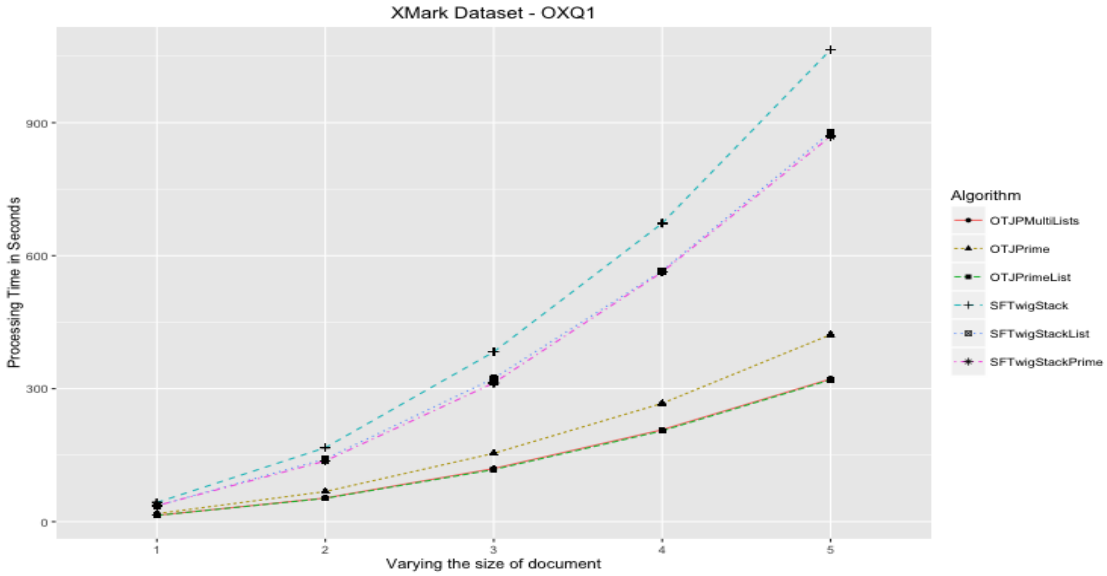
Accordingly, the total number of paired comparisons for the Random dataset can be computed using Formula 6.2 described in Chapter 6 as $= \frac{(6\times(6-1))}{2} \times 7 = 105$. The full results of the pairwise comparisons can be found in Appendix B. The overall results are provided in Table 7.6 which summarises the comparisons to show how many times each algorithm statistically was either faster or slower. As depicted in Figure 7.21, the new algorithms were significantly faster than the other straightforward approaches in most ORTQs except in ORQ_5 where SFTwigStackList had the best performance since the number of intermediate paths is relatively small. It can be seen from the results provided in Table 7.6, OTJPrimeList had the best performance in most cases as it was faster in 30 out of 35 cases while in one case it showed similar performance with its refined version OTJPrimeMultiLists. It was roughly 90, 14 and 22 time faster than the straightforward approaches in ORQ_2, ORQ_4 and ORQ_6, respectively. Interestingly, OTJPrime was the second best algorithm in performance for the Random dataset which is similar to that observed over the TreeBank dataset. when the ordered query has an ordering constraint and the P-C edge connects leaf query nodes to their parent as in ORQ_6, OTJPrimeMultiLists outperformed OTJPrimeList by avoiding scanning irrelevant levels in the buffering lists. For every U test, the effect size suggested that there is a medium to large practical significance. However, when the null hypothesis is rejected, low practical significance was evident.

To conclude, the experiment is used to explore the effects of the CPL relationship and ordered extension because the Random dataset combines structural features of two well-known, real-world datasets, namely DBLP and TreeBank. As a result, the simple buffering technique, OTJPrimeList, showed a superior performance to the other techniques in terms of the number of paths generated and query running time. Moreover, it can be observed that OTJPrime had a better performance than the straightforward algorithms and outperformed the level split buffering algorithm in four queries, namely ORQ_1, ORQ_5, ORQ_6 and ORQ_7.

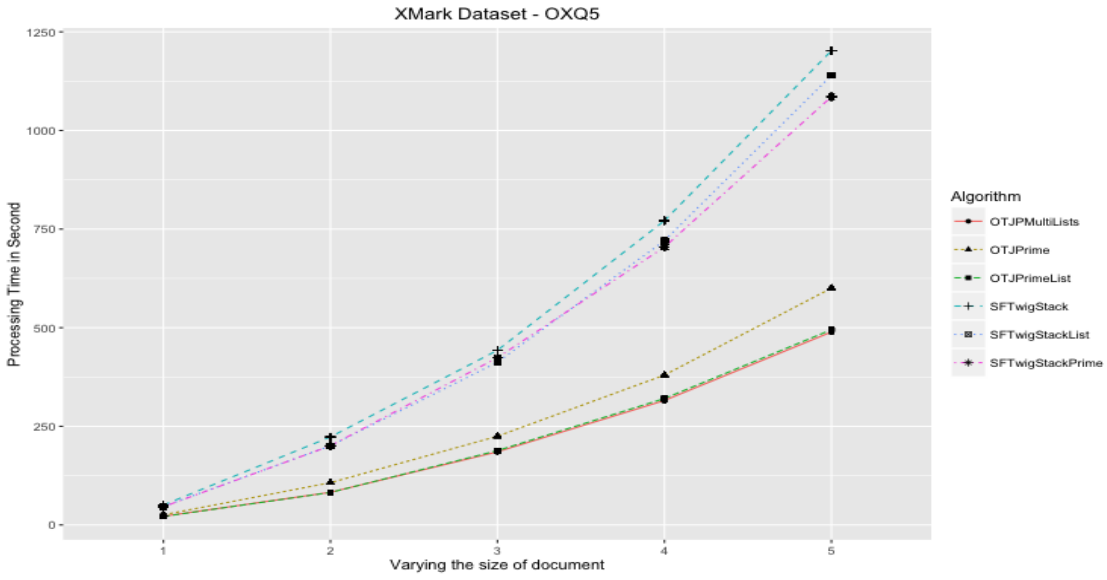
7.4.1.4 Scalability

The experiment aims to simulate and evaluate the scalability of the new algorithms. Two datasets were used, XMark and Random datasets. While the XMark dataset is shallow and data oriented, the random collection has a very recursive structure. Five different versions of XMark were created using the scaling factor from 1 to 5 as described in Section 5.4.4. The Random dataset was partitioned into 10 different datasets to evaluate the scalability of the algorithms over deep, unbalanced trees, see Section 5.4.4. In order to make the experiment more objective, two ordered TPQs were selected over each scalable file of datasets, one of them can be processed by the straightforward algorithms efficiently to some extent when compared to the new approaches. Consequently, OXQ_1 and OXQ_6 were selected for the XMark datasets, and RQ_5 and RQ_7 were chosen to be tested on the Random datasets.

The results for OXQ_1 and OXQ_5 are illustrated in Figure 7.22. From the graphs, it can be observed that the new approaches scaled linearly with the increasing size of the dataset. In addition, with the increase of the size, the benefits of OTJPrime, OTJPrimeList and OTJPrimeMultiLists over the straightforward approaches correspondingly increased.



(a)

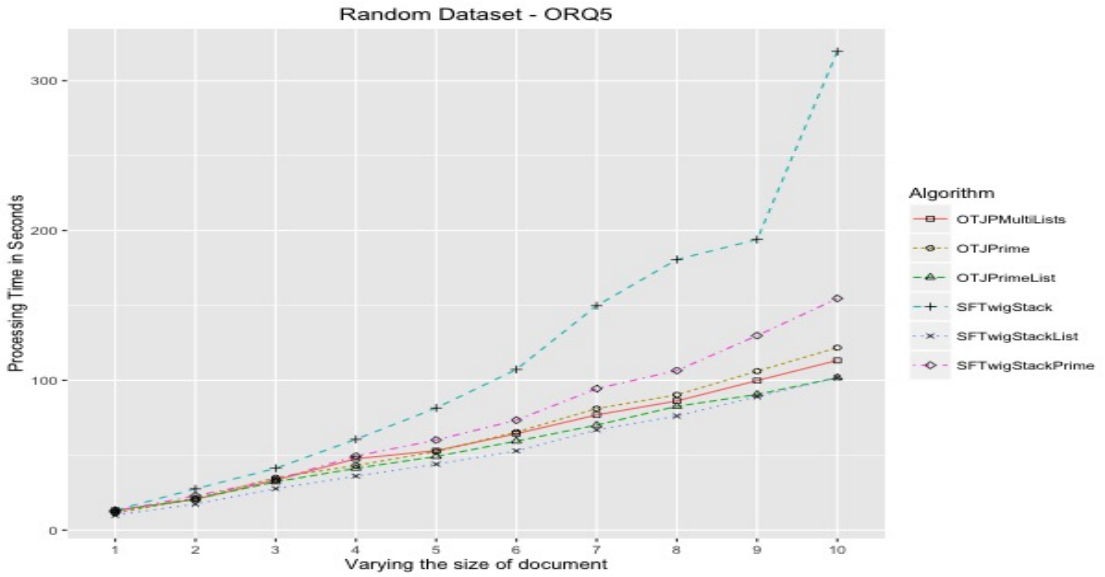


(b)

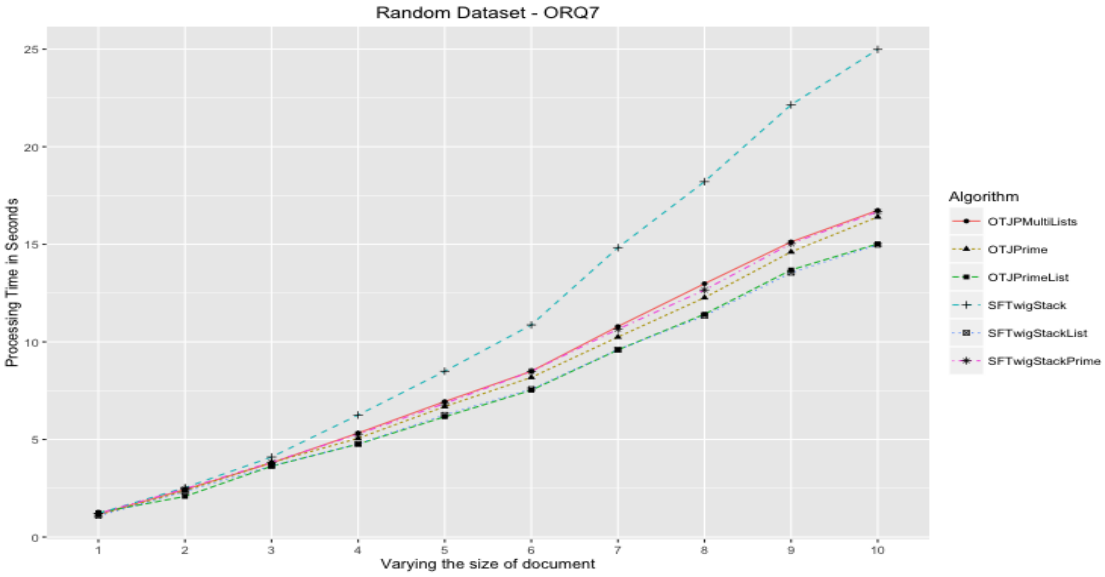
Figure 7.22: Scalability comparison for XMark datasets.

For Random datasets, the scalability results for ORQ_5 and ORQ_7 are depicted in Figure 7.23. Despite the fact that the new approaches generated useless paths for the selected queries, they scaled effectively and presented a linear relationship with the increasing of the size of the dataset. For ORQ_5, all algorithms showed a sub-linear behaviour for growing file sizes except TwigStack. From the data in Figure 7.23b, the performance of the comparable algorithms linearly declined with the increase of the size. Altogether, the experiment considered different structures of ordered TPQs over two groups of different

datasets in terms of structural complexity. It can be, therefore, concluded that the new ordered twig matching algorithms are more scalable in processing large datasets.



(a)



(b)

Figure 7.23: Scalability comparison for Random datasets.

7.4.2 Summary

The experimental results, in the previous sections, have shown that the new ordered algorithms have the ability to ensure pruning of unnecessary elements and in turn enhance runtime efficiency and reduce memory consumption. It can also be observed that the number of path solutions produced by the new approaches was significantly less than the paths generated by the straightforward approaches in all ordered queries tested. It seems possible that the multi list data structure employed by OTJPrimeMultiLists can improve the performance only in one case where the ordered query has leaf query nodes with ordering constraints related to their parents with P-C axes. However, in most cases the extra filtering in OTJPrimeMultiLists had no effect over the original version (i.e., OTJPrimeList); it only causes overheads. The OTJPrimeMultiLists algorithm is only suitable for one class of ordered queries (e.g., ORQ_6), while the OTJPrimeList algorithm can provide better performance for the other classes of ordered queries. When the available memory can not fit elements with ordering constraints (i.e., the runtime memory usage is not bounded to the longest path in the XML tree which is usually small), the OTJPrime algorithm can be used to prune irrelevant elements as they reside in their streams rather than storing them in the main memory. Finally, the scalability tests demonstrated that the new approaches can linearly scale with the increase of dataset size. In summary, OTJPrime and OTJPrimeList algorithms should be used to evaluate ordered TPQs because they have significant performance advantages over the approaches with postprocessing operation.

7.5 Conclusion

In this chapter, the research study presented new approaches that consider the ordered axes and sequence operators incorporated into conventional TPQs. This thesis also introduced novel techniques which can utilise buffering techniques previously proposed in the literature along with the CPL indexing to process ordered TPQs efficiently. Furthermore, the experimental results have shown that the new holistic ordered twig matching algorithms have superior performances to the algorithms with postprocessing in terms of the number of intermediate path solutions and query running time.

In the next chapter, the *TwigStackPrime's* filter strategy will be applied to one-phase holistic algorithms to improve their practical performances by minimising the number of elements stored in the intermediate storage. The property of CPL will be added to the existing advanced preorder filtering functions in the same way it was embedded to the original *getNext* algorithm [40].

Chapter 8

Twig Pattern Matching: Bottom-Up Approach

8.1 Introduction

In the last decade, several twig matching algorithms have been proposed in the literature [40, 5, 144, 147, 146, 89, 87, 185, 130, 132, 128, 22]. In particular, the one phase holistic twig matching algorithms have better practical performance than algorithms performed in two phases such as TwigStack [40] and TwigStackList [144]. This may be due to the fact that the computationally expensive merge join operation is replaced with a simple enumeration process to output the desired query result [20]. Advanced preorder algorithms *getNext, getPart* and *getMatch* are used in one phase algorithms to skip irrelevant elements which are not part of the whole TPQ using weak subtree match filtering (see Definition 4.14). The recursive functions *getPart* and *getMatch* are refined versions of the original *getNext* to improve practical performance.

The *getPart* algorithm [89] was designed to return query nodes in strict preorder instead of a relaxed preorder provided by *getNext*. It also uses efficient mechanisms which forward cursors corresponding to both internal and leaf query nodes. Note that the original *getNext* is efficient only when leaf query nodes are selective because it uses Case 1 with respect to the binary structural relationship, A-D (see Table 6.1 of Chapter 6) to forward cursors of internal query nodes efficiently. In other words, the *getNext* algorithm skips elements conforming to Case 1 of Table 6.1 whereas the *getPart* algorithm discards elements conforming to Case 1, 3 and 4 of Table 6.1 (Case 2 is the matching case). The *getMatch* algorithm extends *getPart* to avoid redundant computations by skipping unnecessary recursive calls. That is, if the internal query node was checked in the previous match, there is no need to recompute the subtree rooted from it since all head elements are guaranteed to be part of its solution. However, these advanced preorder algorithms provide a weak subtree match filtering and for optimal enumeration strict subtree match filtering is required [89, 22]. Therefore, a level split data structure was proposed to enable access to useful child elements below P-C edges and perform strict subtree match checking in

bottom-up holistic twig matching algorithms such as TJStrictPre [89] and GTPStack [22]. As was discussed in Chapter 4, a combination of an advanced preorder function and level split data structure can not filter out leaf query nodes effectively when TPQs with a mix of P-C and A-D edges are processed.

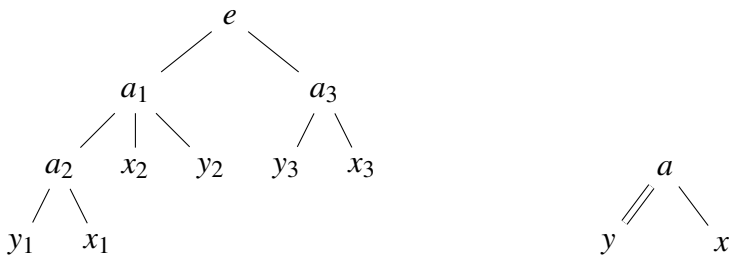
In this chapter, the research study proposes a new solution to avoid unnecessary stack operations used respectively in the TJStrictPre and GTPStack algorithms. The new solution can skip useless elements efficiently as it utilises the Child Prime Label approach devised in Chapter 6. This chapter presents a set of one phase holistic algorithms based on the Child Prime Label approach. The one phase algorithm, TwigPrime is proposed in this thesis to extend TwigFast [132] and TJStrictPre [89] algorithms since they were reported to have the best practical performance among one phase algorithms [89, 88]. An extensive set of experiments is conducted to evaluate the performance, scalability and efficiency of the new bottom-up holistic twig matching algorithms.

The rest of this chapter is structured as follows. Section 8.2 covers some preliminaries including the notation and data structures in one phase holistic algorithms and the limitations of the state-of-the-art bottom-up algorithms. The development of the new algorithm, TwigPrime will be presented in Section 8.3.1. Section 8.4 describes the experimental evaluation and reports the performance comparison between the algorithms. Finally, the chapter will be concluded in Section 8.5.

8.2 Preliminaries

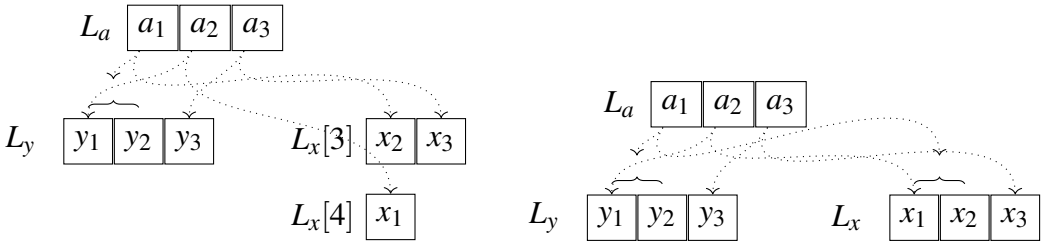
8.2.1 Notation and Data Structure

Most of the notation and data structures used in this chapter are the same as those in Section 7.2.1. In the one phase algorithms, there is one list L_n for each query node n in a TPQ. The only exception is that elements are stored in the intermediate lists (i.e., buffering lists in OTJPrimeList and OTJPrimeMultiLists algorithms) as objects which contain the following information for each element: its label, pairs of intervals for each child query nodes to record positions of children or descendants in their corresponding lists. In order to avoid the use of stacks to control useless elements, each object has two pointers to simulate the layout of nested elements stored in a stack similar to that in TwigStack and TwigStackPrime algorithms explained in Chapter 6. In other words, one pointer represents the element holding the top position in the stack while the second pointer points to the bottom element of the stack. It is thought that the overhead of maintaining pointers is negligible in comparison with the pushing and popping of elements into and out their stacks [132]. The intermediate lists are used to store the final solutions. For each query node, there is a list of elements. Each element e_n in the list L_n has, for each child of n, pairs of start and end pointers pointing to the start and end positions of child and descendant intervals. Start values are updated as elements are processed in preorder while end values are recorded as elements are found not to contribute to any new solution.



(a) an XML tree T_1 (b) Q_1

Figure 8.1: An example to illustrate the child and descendant intervals in bottom-up holistic algorithms.



(a) Level split lists approach (b) Simple lists approach

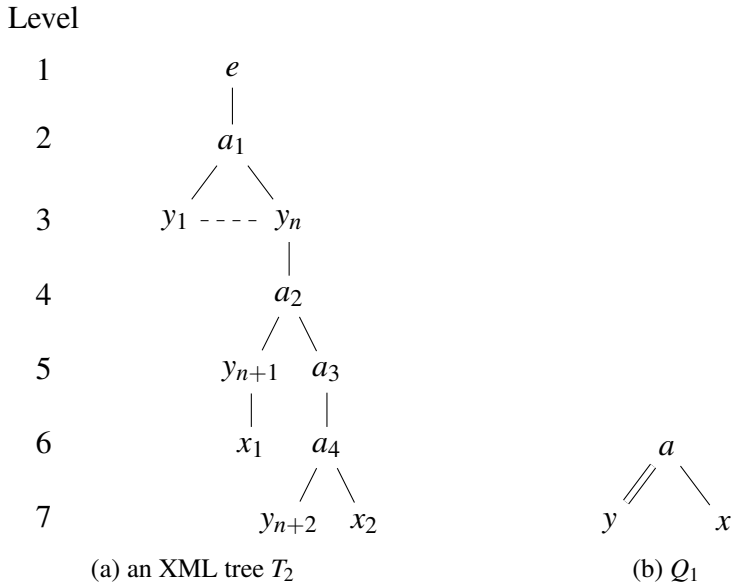
Figure 8.2: Illustration to child and descendant intervals using a level split and simple lists.

Example 8.1. *Consider the XML tree of Figure 8.1 and the query Q_1 = //a[//y]/x. The intermediate storage containing the final solutions with child and descendant intervals is depicted in Figure 8.2a. For each element in L_a, its child x-elements and descendant y-elements can be arranged within minimal intervals. For instance, all y-descendants of a-element a_1 fall into its interval as a_{1start_y} = 1 and a_{1end_y} = 2 so that only relevant y-elements y_1 and y_2 are accessed. Similarly, child x-element of a_1, namely x_1 can be scanned efficiently as a_{1start_x} = 1 and a_{1end_x} = 1 in the relevant level split list which contains x-elements hierarchically nested one level down a_1 (i.e., at level 3 by $L_x[3][1]$). It can be seen from this example, the use of level split lists avoids unnecessary scan of irrelevant elements when enumerating over intermediate lists to output the query results. This due to the overlap of descendants when simple lists are used as shown in Figure 8.2b. With the simple approach introduced in [132], a_1 is associated with x_1 despite the fact that x_1 does not satisfy the P-C relationship with a_1. Eventually, the enumeration algorithm produces matches $(a_1, y_1, x_2), (a_1, y_2 x_2), (a_2, y_1, x_1)$ followed by (a_3, y_3, x_3).*

The following section will discuss the limitations of the up-to-date bottom-up matching algorithms with respect to P-C edges in TPQs. It will also show the importance of strict matching checks in order to guarantee optimal evaluation and enumeration. Simple examples will be presented to demonstrate the potential benefits of using the CPL approach in bottom-up matching algorithms.

8.2.2 Motivations

In this section, the limitations of the state-of-the-art one phase holistic algorithms are addressed in order to improve practical performance. The original TwigFast algorithm was reported [88] to have the best practical performance. The main disadvantage of TwigFast is that it implements only a weak subtree match filtering that inherits from the use of an advanced preorder filtering function, *getNext* of TwigStack [40]. Since a strict filtering is required to provide optimal evaluation, the TJStrictPre algorithm [89] was proposed to extend TwigFast by implementing both strict prefix path and subtree filtering. The prefix path filtering is implemented by means of a chain of stacks similar to that in TwigStack because the use of level split data structure may cause random access if it is implemented naïvely. Furthermore, the strict subtree filtering is performed by using level split lists through extra filtering passes over lists corresponding to inner query nodes. The work of [22] devised a new algorithm, GTPStack which improves TJStrictPre by avoiding the extra filtering passes over the intermediate lists and unnecessary recursive calls of advanced preorder filtering functions. However, these algorithms suffer from a considerable number of useless elements appended to the intermediate lists since they are based on an advanced preorder filtering applying a weak subtree match filtering introduced in TwigStack [40]. It is possible that proposing a new solution to avoid stack operations and skip useless elements using the Child Prime Label approach [11] may improve the overall performance. The following example illustrates how the previous algorithms process TPQs using a combination of preorder and postorder filtering with and without post-processing filtering passes.



(a) an XML tree T_2 (b) Q_1

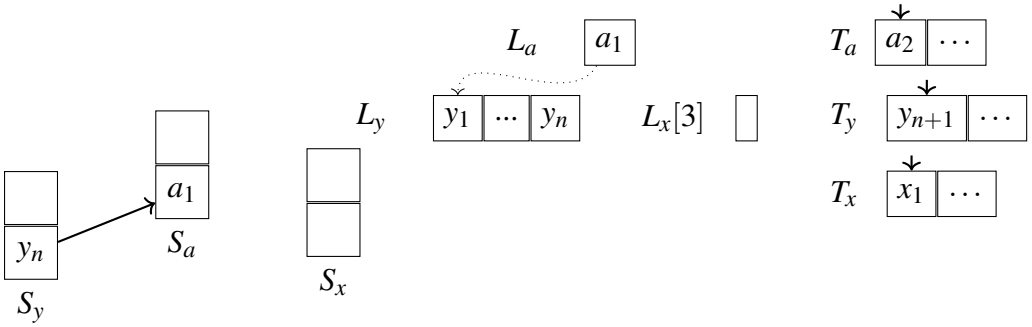
Figure 8.3: Illustration of TJStrictPre and GTPStack algorithms using the level split approach.

Example 8.2. *Consider running TJStrictPre to process the query Q_1 = //a[//y]/x against the XML tree T_2 of Figure 8.3. Initially, the algorithm associates the x-node with level*

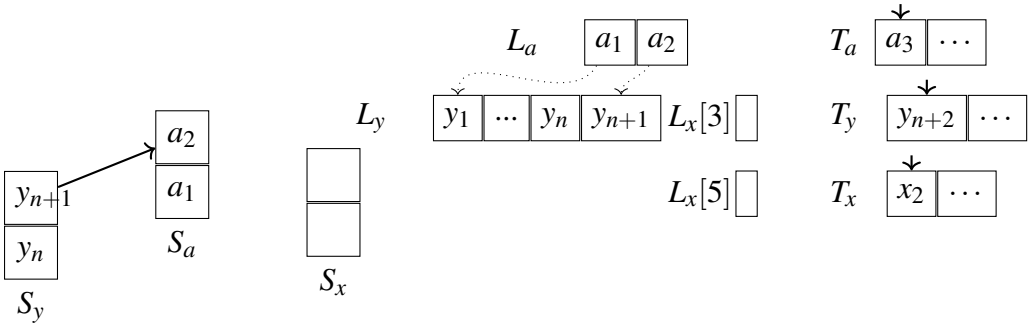
split lists in order to avoid the overlap of descendants and provide optimal enumeration. The TJStrictPre algorithm in contrast to TwigFast uses one stack S_{q_i} for each query node q_i to enable strict prefix path filtering. That is, the current query node is used to clean the closest ancestor stack by popping off non-contained elements (i.e., reference objects in the TJStrictPre algorithm to avoid significant cost incurred from removing useless elements). From the example in Figure 8.4, elements a_1 to a_3 are useless because they fail subtree filtering with respect to x-node after checking their corresponding level split lists when streams reach the end after reading x_2 in Figure 8.4c. However, The TJStrictPre algorithm stores them in the intermediate list of query node a. The TJStrictPre algorithm relies on extra pass filtering for intermediate lists corresponding to inner query nodes to perform subtree filtering checks in order to provide optimal enumeration as shown in Figure 8.4d. Elements a_1 to a_3 are removed from L_a since they do not have proper child x-elements in $L_x[3]$, $L_x[5]$ and $L_x[6]$, respectively. It should be noted that x_1 is not added to the intermediate storage of x-node because it fails strict prefix path matching with a_2 as the top element of S_a. With respect to the GTPStack algorithm, Figure 8.5 depicts operations of a combination of preorder and postorder checks during the query processing of Q_1 over the XML tree T_2 in Figure 8.3 by the GTPStack algorithm. Compared with the TJStrictPre algorithm, the GTPStack algorithm does not append elements to the intermediate storage without performing strict subtree checks. Elements from a_1 to a_4 are nested in S_a and when streams reach the end, only a_4 is stored in the intermediate storage because a_1 to a_3 do not satisfy strict subtree checks. To avoid storing unordered elements in the intermediate storage because of the relaxed pop order used by the GTPStack algorithm as described in Chapter 4 (see Section 4.2.3); a linked list is used as the main intermediate storage structure instead of a dynamic array. As a result, each item in a stack is represented with a pair (prev:next) to enable concatenation of elements according to the document order in a constant time. Figure 8.5c illustrates this idea when y_n is appended to the intermediate storage after its descendants y_{n+1} and y_{n+2}. This shows how GTPStack avoids producing a post-order list of elements for each query node as in TwigList by using a list concatenation even though Arrays have better cache locality as compared to linked lists.

However, all the previously mentioned algorithms suffer from some serious drawbacks. A shortcoming of both TJStrictPre and GTPStack is that they store irrelevant elements in the intermediate storage. One of the limitations with TJStrictPre is that it relies on extra filtering pass to avoid pushing many irrelevant elements and guarantee linear output enumeration with respect to the result size. A serious weakness with this GTPStack, however, is that it focuses on enabling combined filtering checks before storing elements in the intermediate storage achieved by a list concatenation which may occasionally result in performance degradation. A possible explanation for this might be that a positional access which is excessively performed during the output enumeration requires O(n) in a linked list and O(1) in a dynamic array where *n* is the number of elements stored [88, 22, 153, 215].

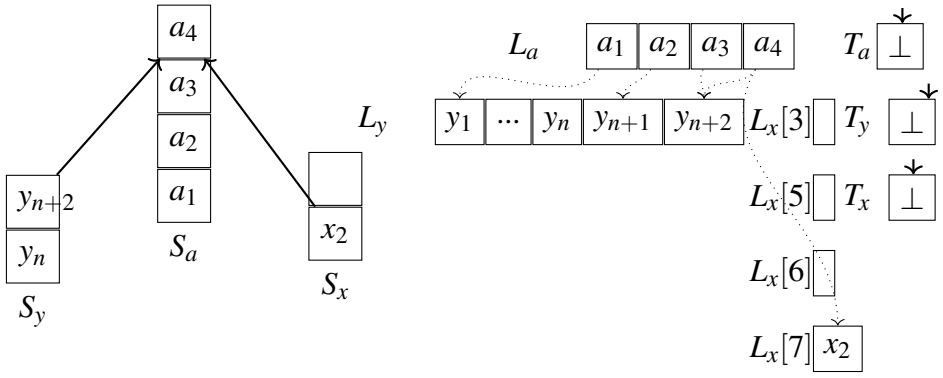
The next section presents a new one phase holistic twig matching algorithm which combines the efficient selection of useful elements for TPQs with a mix of P-C and A-D edges introduced in Chapter 6 and the level split data structure for storing intermediate results without using stacks in order to enable strict prefix checks. In addition, the CPL approach will be introduced to the advanced preorder filtering functions used by the TJStrictPre and GTPStack algorithms, namely *getPart* and *getMatch*, respectively, in the same way that *getNext* of TwigStack was augmented with the CPL approach.



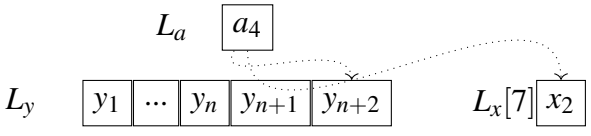
(a) When y_n has been returned by the filtering function.



(b) After x_1 has failed the strict prefix matching.

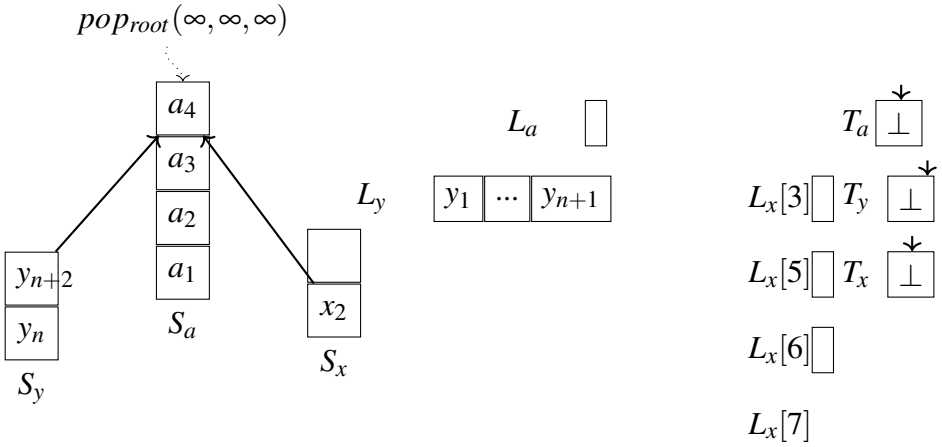


(c) After processing x_2 and filtering phase is terminated.

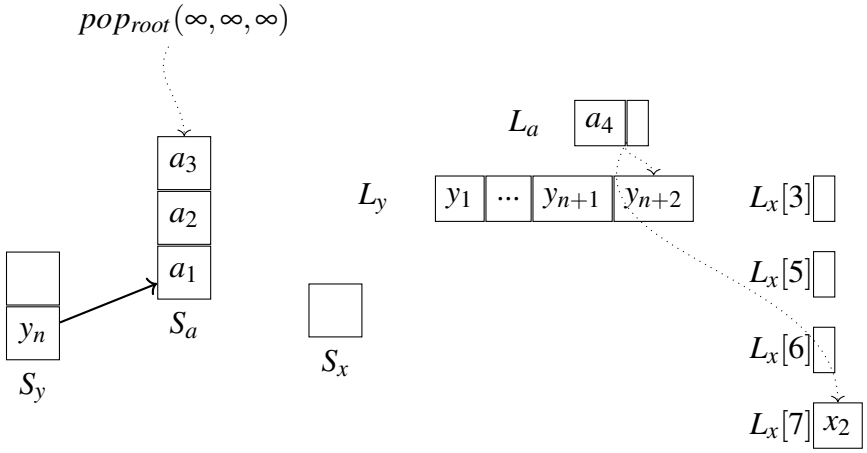


(d) Extra pass filtering over the list of a-elements to provide optimal enumeration.

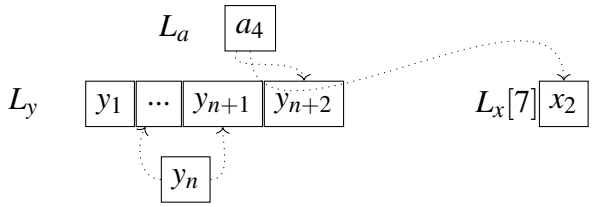
Figure 8.4: Illustration of TJStrictPre evaluating Q_1 on T_2 of Figure 8.3.



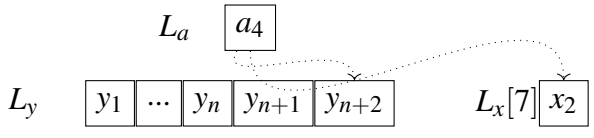
(a) After processing x_2 and filtering phase is terminated.



(b) When a_4 and all its descendants are poped out from their stacks.



(c) An example to illustrate the use of linked lists in the GTPStack algorithm.



(d) The intermediate storage prior to the output enumeration.

Figure 8.5: Illustration of GTPStack evaluating Q_1 on T_2 of Figure 8.3.

8.3 Bottom-Up Twig Matching Algorithm with Child Prime Label

In this section, the thesis introduces a set of new bottom-up holistic twig matching algorithms which combine the advantages of the previous approaches [132, 89, 22, 11], TwigPrime and its versions are modification of TwigFast [132] which combine the efficient selection of useful elements for TPQs with a mix of P-C and A-D edges introduced in [11] (see Chapter 6) and utilises level split data structure as the main intermediate storage. They further improve TwigFast by strictly checking prefix path matching for P-C relationships before storing the intermediate results. Moreover, the state-of-the-art filtering strategies *getPart* and *getMatch* will be extended to apply the CPL approach in order to explore the potential benefit of the CPL approach in the contemporary one phase holistic algorithms.

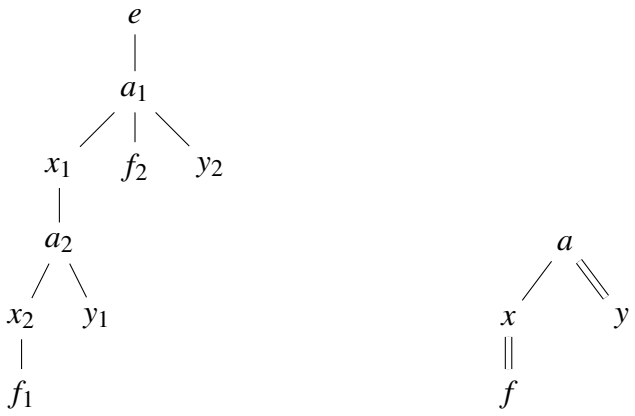
8.3.1 Bottom-Up Twig Matching Algorithm: TwigPrime

The new approach can be seen as an alternative to the TwigFast algorithm. The original TwigFast remains the same with the differences being in the advanced preorder function *getNext* which is based on the CPL approach and the use of a level split data structure to store the intermediate results. The use of pointers in TwigPrime and its refined versions is similar to that in [132] (see Section 8.2.1).

The structure of the main algorithm, TwigPrime presented in Algorithm 11, is a total rewrite of TwigFast, and more complex than the original algorithm. In [132], there is one list containing matches for each query node, the list is sorted in preorder. Each element in the list has a recorded interval for each child query node. Interval start values are recorded as elements are appended to intermediate lists while interval end positions are recorded when elements can not be part of any further match. In order to avoid the use of stacks and construct intervals effectively, each element appended to the list has a pointer to the closest ancestor in the same list. There is also a tail pointer associated with each list which indicates the candidate for the parent query node. An advantage of this approach is that there is no overhead for maintaining a set of stacks. However, it fails to give sufficient consideration to prefix path filtering checks so that elements are added without having relevant parents. Note that using the level split data structure as the main intermediate storage is hard to transfer directly to TwigFast unless some query nodes are treated carefully. For illustration, Figure 8.7 shows the intermediate results after running TwigFast to process Q_1 against T_2 in Figure 8.6. When f_2 is returned, the algorithm can record interval end positions for x_1 and x_2 since the tail for x-elements points to x_2 and x_2 has an ancestor pointer to x_1 in the same list as depicted in Figure 8.7b. Assuming the level split approach is used as shown in Figure 8.7a, if x_2 is only pointed to by the tail pointer, then one match including x_1 may be missed. Accordingly, to set descendant intervals correctly, there must be a tail for each level and each tail must be checked separately. In this thesis, a new procedure is introduced which presents the novel level split filtering

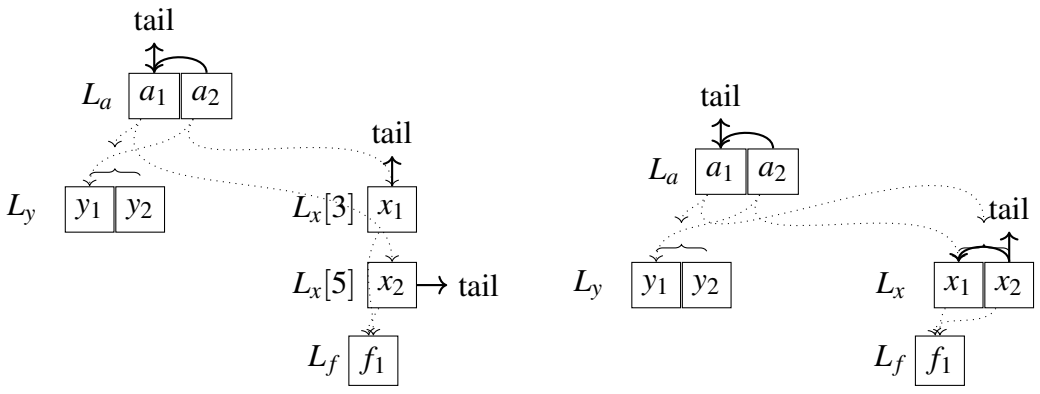
used by the TwigPrime algorithm without the need for stacks. The level split tail filtering represented by Algorithm 12 is mainly based on considering one type of query node which plays an essential role in the matching process in TwigPrime. Generally, there is only one situation that the level split tail filtering happens. That is, if the incoming element for a query node q_n has an A-D relationship to the parent query node q_p that has, in turn, P-C relationship with the parent query node, the tail for every level split list corresponding to the query node q_p must be checked to record the end positions correctly. This definition can be formalized as in Definition 8.3. It should be also noted that using a pointer to the closest ancestor in the same list is unnecessary when an element has P-C relationship to the parent as they are stored in different lists. Henceforth, a tail pointer is sufficient to track potential parents or ancestors for query nodes under P-C edges.

Definition 8.3 (AD Follows PC)**.** *If and only if given a query node p, which is connected with P-C edge to its parent, and its A-D child c, suppose n separate level split lists of p has been visited. In intermediate lists of p, all elements which are pointed by n tails will be checked. The tailed elements that are not ancestor of the current element e_c will be assigned their end interval values.*



(a) an XML tree T_2 (b) Q_1

Figure 8.6: An example to illustrate tail pointers for level split data structure.



(a) Nested tail checks for level split approach (b) Straightforward tail checks, TwigFast

Figure 8.7: Intervals for intermediate result handling approaches after processing f_1.

Algorithm 11: TwigPrime

Input: A TPQ Q

// initialization
// initialise $L_{n_i} = \emptyset$ for each $n_i \in TPQ\ Q$ if n_i is root or $n_i \in childrenAD(parent(n_i))$ and $n_i.tail$ = -1
// initialise an array of $L[]_{n_i} = \emptyset$ for each $n_i \in TPQ\ Q$ if $n_i \in childrenPC(parent(n_i))$
while $\neg$*end(getRoot(Q))* **do**
    q_{act} = getNext(getRoot(Q)) // see Algorithm 5
    v_{act} = getElement(q_{act})
    if $\neg$ *isRoot*(q_{act}) **then**
        setEndPointerParent(q_{act},parent(q_{act})) // see Algorithm 12
    if *isRoot*(q_{act}) $\vee$ *getParentTail*(q_{act}) $\neq$ *-1* **then**
        if $q_{act} \in$ *childrenPC(parent(qact))* **then**
            h = level(v_{act}) - 1 // parent should be stored one level higher
            v_p = getVectorElement(parent(q_{act}),h)
            if $\neg$ *PCrealtionship*(v_p,v_{act}) **then**
                // here to perform strict prefix path filtering which TwigFast misses
                advance(q_{act})
                continue // skip the following lines and moves to the next cycle
        if $\neg$ *isLeaf*(v_{act}) **then**
            // set the end values for all elements in $L_{q_{act}}$ which are not ancestor of v_{act}
            // $\forall\ n_i \in$ children(q_{act}) $v_{act}.start_{n_i}$ = length(getVector(n_i))
            // $v_{act}.ancestor$ = getTail(q_act) // pointer to the closest ancestor or -1 if it does not have one
            // setTail(q_{act}) to length(getVector(q_{act})) // this to set the tail pointing to v_{act} as the open element for this list
        // append v_act to the corresponding list
    advance(q_{act})
setResetEndPointers(getRoot(Q)) // process remaining open elements using ∞
extraPassFiltering(getRoot(Q)) // clean intermediate results with postorder checks introduced in [89]
enumerateResults()
Function `getVector(`*Query node q,Integer level*`):`
    // return the regular intermediate result list if q is below an A-D edge or split list given by level if q is below a P-C axis.
Procedure `setResetEndPointers(`*Query node q*`):`
    if $\neg$*isLeaf(q)* **then**
        // set th end values for all remaining elements pointed by tails not equal to -1 and tails' ancestors (if any)
        // if q has P-C, then $\forall$ level $\in$ used level of $L[level]_q$ set th end values for all remaining elements pointed by tails not equal to -1
    $\forall\ n_i \in$ children(q) setResetEndPointers(n_i)
Procedure `extraPassFiltering(`*Query node q*`):`
    // read inner query nodes in postorder and any element which does not satisfy the postorder filtering will be deleted. After that, the intermediate list will be resized. Interval pointers for parent query nodes will be updated according to the change in their child lists.

Algorithm 12: Level split tail filtering

```
1  Function isADfollowsPC(Query node q):
2      p = parent(q)
3      if q ∈ ChildrenAD(p) then
4          if p ∈ ChildrenPC(parent(p)) then
               return : true
       return : false
5  Function getTail(Query node q,Integer h):
6      if isRoot(q) ∨q ∈ childrenAD(parent(q)) then
           return : q.tail
7      else
           return : q[h].tail
8  Function getParentTail(Query node q,Query node p):
9      h = level(getElement(q))
10     if ¬ isADfollowsPC(q) then
           return : getTail(p,h-1)
11     else
12         ∀ level ∈ used level if getTail(p,level) ≠ -1) then
               return : getTail(p,level)
           return : -1
13 Procedure setEndPointerParent(Query node q,Query node p):
14     if isADfollowsPC(q) then
15         ∀ level ∈ used level of L_p
16         if getTail(p,level) ≠ -1) then
17             v_act = getVectorElement(p,level)
18             if getEnd(v_act) < getStart(getElement(q)) then
19                 markEnd(v_act) // set the end valuse for ecah n_i ∈ children(p)
20                 // v_act.end_{n_i} = length(getVector(n_i)) -1
21     else
22         while getParentTail(q,p) ≠ -1 do
23             v_act = getVectorElement(p,0)
24             if getEnd(v_act) < getStart(getElement(q)) then
25                 markEnd(v_act) // set the end values for each n_i ∈ children(p)
26                 setTail(v_act) // set tail for the particular query node.
27             else
28                 break
29 Function getVectorElement(Query node q,Integer level):
30     // return the current element pointed by the tail of the regular intermediate result
         list if q is below an A-D edge or split list given by level if q is below a P-C axis.
31 Procedure setTail(Query node q,Integer level):
32     // set the tail to point to the closest ancestor of the current tail if any exists,
         otherwise -1. if q is below an A-D edge q.tail or q[level].tail given by level if q
         is below a P-C axis.
```

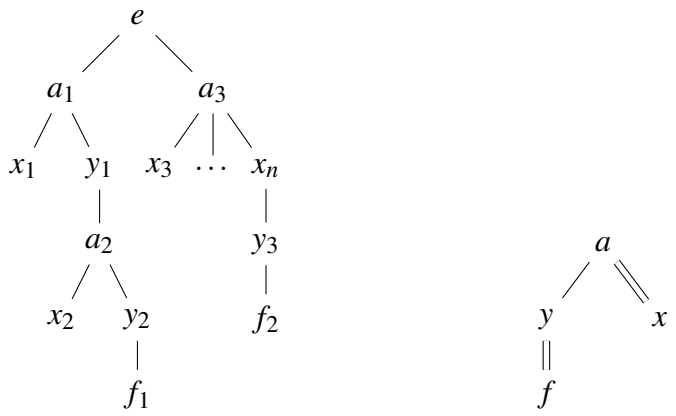
Moving on now to the main algorithm of TwigPrime, Algorithm 11 presents the general framework for constructing intermediate results in preorder manner, extending TwigFast [132]. It supports any combinations of preorder and postorder filtering and simple or level split vectors. It also can be extended to use advanced preorder filtering functions such as *getPart* and *getMatch* since elements are stored in preorder. It iteratively invokes *getNext* based on the CPL approach [12] to identify the next query node for processing. As opposed to the original TwigFast algorithm, elements are passed directly to the intermediate result if they have passed a strict prefix path filtering (Lines 10-16), If the head element of q_{act} fails to satisfy the strict prefix path matching, its cursor is shifted to point to the next element in the stream and the algorithm proceeds to the next iteration. Prior to that, Line 8 performs a weak prefix matching (see Definition 4.13) by determining the end positions for elements which are not ancestors of the head element of q_{act} in intermediate lists corresponding to $parent(q_{act})$ according to Definition 8.3 when level split vectors are used to avoid false negative errors. This is performed though calling the procedure *setEndPointerParent* in Algorithm 12. After that, if the head element of q_{act} has the ancestor extension and q_{act} is not a leaf query node, Line 18 updates the end values for elements which are not ancestor if the current element in the same list. Then, the start positions for intervals of element v_{act} will be determined which are equal to the current lengths of v_{act}'s children lists. After that, the tail and ancestor pointers are updated. The purpose of these pointers is to identify elements which still have potential descendants. For example, in Figure 8.7, when y_2 is considered as the head element of $q_{act} = y$, it indicates that a_2 will not have any further descendant so that the end positions intervals for a_2 are recorded. At Line 22, the current element is appended into the corresponding list. The cursor of the current query node q_{act} is forwarded and the algorithm proceeds to the next cycle. When streams are ended the algorithm terminates the top-down processing by using the largest range-based label (∞, ∞, ∞) to enforce all open elements to update end positions for their intervals. To perform strict subtree checks, the intermediate results are filtered bottom-up in the post-processing procedure *extraPassFiltering* at Line 25. An internal element e_q is removed from the list L_q if and only if for any $n_i \in children(q)$, $e_q.start_{n_i} > e_q.end_{n_i}$. Finally, once the intermediate storage contains elements with their intervals, TwigPrime will enumerate the output based on the enumeration algorithm introduced in [185] and extended in [89, 22] to use child intervals when the level split approach is applied.

The following example demonstrates the effect of TwigPrime in filtering useless elements based on a combination of the CPL level split approach without consuming extra storage, manipulating node processing order and using stacks, see Section 4.2.3.

Example 8.4. *Consider the XML tree T_2 of Figure 8.8, and the TPQ Q_2 = //a[/y//f]//x in Figure 8.8. The first call of getNext() in TwigPrime returns a and a_1 is the current element v_{act} because it satisfies the descendant extension condition and the CPL relationship regarding y-node. Since query node a is not a leaf, the list is checked to identify elements which are not ancestors of a_1 in order to set their end positions of their child intervals.*

The lists are empty so that the procedure does not do anything. Next the closest ancestor of v_{act} is set to -1. After that, the start position of a_1's descendant interval for x-node is set to 0 as $length(L_x) = 0$ whereas the start position of a_1's child interval for y-node is set to 0 as $length(L[3]_y) = 0$. Now the tail of query node is pointed to a_1 as $length(L_a) = 0$, and a_1 is appended to the simple list L_a and the cursor of T_a is forwarded to a_2. TwigPrime proceeds to the next cycle and the second call of getNext() returns x_1. The procedure $setEndPointerParent(x,a)$ is called to perform a weak prefix path filtering. Since a_1 is the tail for query node a and x_1 is contained by a_1, the procedure finishes without performing any thing, T_x is pointed to x_2. After that, y_1 is returned by getNext(), a weak prefix path matching is performed through calling $setEndPointerParent(y,a)$. Since y-node has P-C edge with a-node, a strict prefix path matching is performed through Lines 10-16. After finding that y_1 satisfies P-C relationship with a_1, the start position of y_1's descendant interval for f-node is recorded, and y_1 is appended to the corresponding level split list $L[3]_y$. Now a_2 is considered and returned to the main algorithm, because $a.tail = 0$ and a_1 is ancestor of a_2, $a_2.ancestor = 0$, and $a.tail = 1$. Then, the start position of a_2's descendant interval for x-node is set to 1 as $length(L_x) = 1$ while the start position of a_2's child interval for y-node is set to 0 as $length(L[5]_y) = 0$. x_2 and y_2 are appended to their corresponding intermediate lists because they satisfy weak and strict prefix path filtering, respectively. After that, f_1 is stored in the list L_f, and TwigPrime proceeds to the next iteration. TwigPrime skips a_3 since it does not satisfy the CPL relationship that is CPL of a_3 is not divisible by the prime number associated with the tag name y. The algorithm terminates the inner cycle and finishes the top-down processing to construct the child and descendant intervals through calling $setResetEndPointers(a,\infty)$ to set the remaining end values. That is, the end positions of a_2's intervals are set to $a_2.end_x = 1$ and $a_2.end_y = 0$, respectively. Similarly, $a_1.end_x = 1$ and $a_1.end_y = 0$. Finally, the end values for y_1 and y_2 are set to 0 as they both are ancestors of f_1 (i.e., $length(L_f) - 1 = 1 - 1 = 0$). In Line 25, a strict subtree filtering is performed to provide optimal enumeration by checking for non-empty child and descendant intervals. In this example, no element is removed because of the use of the CPL approach in TwigPrime which avoids the process of a_3. After that, the output is enumerated according to the enumeration algorithm used in [185, 89, 22]. As a result, the query result consists of three matches (a_1,x_1,y_1,f_1), (a_1,x_2,y_1,f_1) and (a_2,x_2,y_2,f_1). Figure 8.9 depicts construction of intervals during the query processing of Q_2 in different bottom-up algorithms and shows the number of elements stored by each algorithm. From Figure 8.9 it can be seen that only TwigPrime was successful in filtering out irrelevant elements to Q_2.

The improvement of TwigStackPrime [12] can trivially be ported to the state-of-the-art algorithms such as TJStrictPre, TJStrictPost and GTPStack. In this thesis, TJStrictPrePrime, TJStrictPostPrime and GTPStackPrime, which are less involved modifications of the original algorithms, are proposed. This is due to the fact that they are based on advanced preorder filtering strategies (i.e., *getPart* and *getMatch*) which are extensions of *getNext*.



(a) an XML tree T_3 (b) Q_2

Figure 8.8: An example to illustrate the basic notations of TwigPrime.

To achieve the above improvement, *getPart* and *getMatch* are augmented with *getElement* introduced in Chapter 6 to improve structural relationship checks using the CPL approach. As a result, *getPart(q)* and *getMatch(q)* return an element e_q of a query node $q \in TPQ$ with four properties:

i e_q has a descendant element e_{q_i} in each of the streams corresponding to its child elements where e_{q_i} is the head element of a query node q_i = *children(q)*.

ii each of its child elements satisfies recursively the first property.

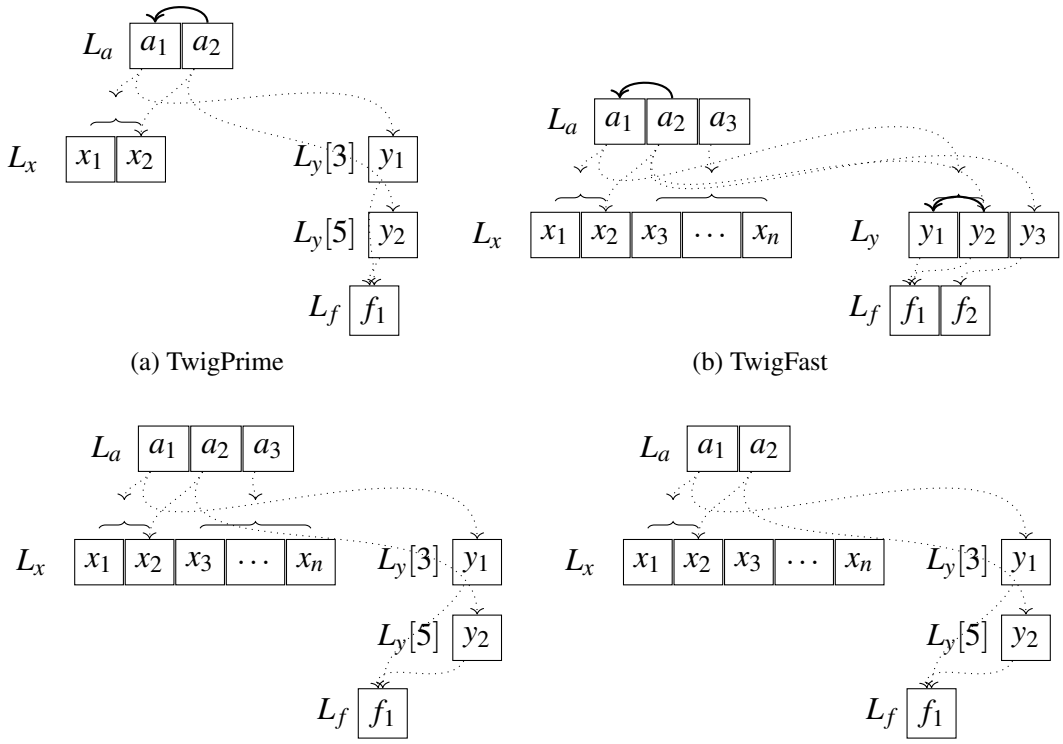
iii if q has Parent-Child edge(s) with its child query nodes, then e_q has a child e_{q_i} in T_{q_i} for each query node q_{q_i} = childrenPC(q) (this property is checked in *getElement* function).

iv if $\neg isRoot(q)$, then e_q has a relevant ancestor e_p stored in the main algorithm which has been the head element of a query node p = parent(q) in previous calls of *getPart(p)* (*getMatch(p)*, respectively).

In the same way, TwigPrimePart and TwigPrimeMatch are proposed as refined versions of TwigPrime to use the *getPart* and *getMatch*, respectively. The following section shows the correctness of the new approaches.

8.3.1.1 Analysis of TwigPrime

This section shows the correctness of the new algorithms and analyses their complexities. The correctness of TwigPrime, TwigPrimePart and TwigPrimeMatch follow from the correctness of the level split tail filtering discussed in Section 8.3, the correctness of preorder filtering used in TwigStackPrime [12] described in Section 6.4.2, and the correctness of the TwigList output enumeration algorithm [185] which is trivially extended to use child intervals when elements are stored in level split lists as in [89]. The correctness of TJStrictPrePrime and TJStrictPostPrime follows from the correctness of preorder filtering used in TwigStackPrime [12] described in Section 6.4.2, and the correctness of TJStrictPre



(a) TwigPrime

(b) TwigFast

(c) TJStrictPre without extra pass filtering

(d) TJStrictPre with extra pass filtering and GTP-Stack

Figure 8.9: Bottom-up algorithms and their corresponding intermediate storages for processing Q_2 against T_3 in Figure 8.8.

[89] and TJStrictPost [89], respectively. Similarly, the correctness of GTPStackPrime follows from the correctness of preorder filtering used in TwigStackPrime [12] described in Section 6.4.2, and the correctness of GTPStack [22].

Lemma 8.5. *Let e_q be an element corresponding to the query node q in the intermediate storage. Then its child and descendant intervals are correctly recorded.*

Proof. Query node q is either leaf or internal. If q is a leaf query node, the lemma holds. Otherwise, it is known by Lemma 6.18 e_q is returned by *getNext* because it satisfies the properties 1 and 2a (see Chapter 6). Therefore, e_q is appended into the intermediate list before child and descendant elements of e_q are stored in their corresponding lists, and the start positions of the intervals thus can be set correctly at Line 19 of TwigPrime. Using Lemma 6.21, all elements in the XML tree which are part of some solutions at subtree rooted at e_q will be returned in preorder. Henceforth, all child and descendant elements of e_q are stored in the intermediate storage while e_q is pointed by the tail of q. Using lemma 6.20, the procedure *setEndPointerParent* correctly records the end values for e_q's intervals. For both cases the lemma holds. □

Using the above lemma and the lemmas introduced in Section 6.4.2, the next theorem will be used to prove the correctness of TwigPrime, TwigPrimePart and TwigPrimeMatch.

Theorem 8.6. *Given a twig pattern query Q and an XML document D, Algorithms TwigPrime, TwigPrimePart and TwigPrimeMatch correctly construct the intermediate results of Q on D.*

Proof. In Algorithm *TwigPrime*, *getNext(root)* is repeatedly invoked to determine the next query node to be processed. Using Lemma 6.18, it is known that all elements returned by $q_{act} = getNext(root)$ have the child and descendant extension. If $q_{act} \neq root$, Line 8, the algorithm sets the end values for all elements in the intermediate lists $L_{parent(q_{act})}$ that are not ancestors of the head element of q_{act} by Lemma 6.21. After that, it is already known q_{act} has a child and descendant extension so that Line 9 checks whether the tail of $parent(q_{act})$ is pointed to proper ancestor or not. If so, it indicates that it does not have the ancestor extension, and it can be discarded safely to continue with the next iteration. Otherwise, the current head element of q_{act} has both the ancestor and child and descendant extensions which guarantee its participation in a weak match of prefixed path from itself to the root. After that, if q_{act} is connected to the parent query node with P-C edge, Lines 10-16 ensure that the current element has a strict match of a prefixed path. If the head element fails to pass a strict prefix path filtering, then it can be skipped safely to proceed to the next cycle. Otherwise, the corresponding list of v_{act} is cleaned by setting end values of intervals for elements which do not contain the head of v_{act}, and the start positions of intervals for v_{act} are recorded, using Lemma 8.5. Then, if v_{act} has an ancestor in the same list, the ancestor pointer of v_{act} is pointed to $q_{act}.tail$. Otherwise, $v_{act}.ancestor$ is set to -1 indicating that it does not have a proper ancestor in the list. Finally, $q_{act}.tail$ is updated to point to v_{act}, and v_{act} is pushed into its corresponding intermediate list. Once the intermediate storage containing elements with their intervals correctly set, it is straightforward to perform the output enumeration. □

The correctness holds for TPQs with both Ancestor-Descendant and Parent-Child relationships. In addition, TJStrictPrePrime, TJStrictPostPrime and GTPStackPrime are correct due to the correctness of the preorder filtering used in TwigStackPrime [12] described in Section 6.4.2 and the correctness of the original algorithms introduced in [89, 22].

With respect to the run time and space complexity of the proposal algorithms, the new algorithms reads elements form data streams only once in a single forward scan through advanced preorder filtering functions. When elements are appended to the intermediate storage, each child check and interval set takes constant time. Therefore, the worst-case time and space complexity for building up the intermediate storage is $O(f \times |Input|)$ where f is the maximum fanout at any query node in TPQ with n query nodes and Input is the sum of the lengths of the n input lists. It can thus be suggested that the new approaches guarantee optimal evaluation for the case where the TPQ has Ancestor-Descendant edges or there are only Parent-Child edges connected the leaf query nodes, similar to that provided by TwigStackPrime. Thus, elements are only stored in the intermediate result if they contribute to the final result. Therefore, the intermediate result can be enumerated in linear

time $O(n \times |Output|)$ where Output is the number of twig matchings. However, in the case where P-C axes connects internal query nodes, linear performance for output enumeration can be achieved by performing a strict subtree filtering (i.e., extraPassFiltering at Line 25), but the algorithms can not guarantee optimal evaluation. In other words they can provide optimal enumeration (i.e., all elements in internal lists must be part of the final result). Consequently, the worst-case I/O and CPU time complexity is linear with respect to the sum of the input list sizes and the size of the output result. For example, to demonstrate the difference between optimal evaluation and enumeration, TwigPrime in Figure 8.9a guarantees optimal evaluation while TJStrictPre and GTPStack of Figure 8.9d provides optimal enumeration, and both TJStrictPre without a strict subtree matching and TwigFast show suboptimal evaluation in Figures 8.9c and 8.9b, respectively. The space complexity of the new approaches is $O(|Input|)$ which is linear with respect to the total number of elements whose tags appear in TPQs. This is due to the fact that they directly construct the intermediate results. However, when the new algorithms are optimal, the $\Omega(u)$ lower bound is matched, where u is the total number of elements to which query nodes can be matched (i.e., optimal evaluation) [198, 88]. However, an early enumeration approach introduced in [53] can significantly reduce the intermediate storage size needed to return query answer. The early enumeration starts when the incoming element corresponding to the first branching query node does not have a relevant ancestor in the corresponding intermediate list or stack of the first branching query node.

The next section describes the experiments to evaluate the performance of the new algorithms and test the research hypothesis introduced in Section 4.3.2.

8.4 Experimental Evaluation

The following experiments explore the effects of the CPL approach, different advanced preorder filtering strategies and different intermediate storage approaches in bottom-up holistic twig matching algorithms. Hence, this section provides the experimental results of the performance comparison of the new bottom-up twig matching algorithms, namely TwigPrime, TwigPrimePart, TwigPrimeMatch, TJStrictPrePrime, TJSTrictPostPrime and GTPStackPrime against state-of-the-art holistic algorithms: TwigList [185], TwigFast [132], TJStrictPre [89], TJStrictPost [89] and GTPStack [22], with significantly different XML datasets. To improve the efficiency of the output enumeration, TwigList and TwigFast are incorporated with the strategy of next sibling links introduced in [185]. With the exception of TwigList and TwigFast, the algorithms in the experiments are implemented to use level split approach as the default setting unless they are coded with "_" as suffix to indicate intermediate results are stored in simple lists (e.g., TwigPrime_). When TwigPrime, TwigPrimePart, TwigPrimeMatch TJStrictPrePrime, TJSTrictPostPrime and GTPStackPrime use the simple list approach, they are incorporated with the next sibling links. Moreover, the following simple notation is used, where TwigPrime_N, TwigPrimePart_N and TwigPrimeMatch_N stand for TwigPrime, TwigPrimePart and TwigPrimeMatch utilise the simple list approach and the strategy of next sibling links, respectively. Throughout this section, the term "CPL" refers to bottom-up holistic algorithms based on the CPL approach combined with the level split, while the term "CPL_" refers to bottom-up holistic algorithms based on the CPL and simple list approaches. As a result, the "CPL" includes TwigPrime, TwigPrimePart, TwigPrimeMatch, TJStrictPrePrime, TJSTrictPostPrime and GTPStackPrime algorithms. The "CPL_" refers to the set of algorithms including TwigPrime_, TwigPrimePart_ TwigPrimeMatch_, TJStrictPrePrime_, TJSTrictPostPrime_. Note that the CPL approaches are used to denote bottom-up twig matching algorithms using the CPL relationship introduced in Chapter 6 regardless the approach used to store intermediate results. Similar to the experiments in Chapter 6, all the algorithms tested in the experiments were implemented and added to the query processor introduced in Section 5.2.2.2, and versions of algorithms are implemented as new algorithms to make sure the overhead of the complex algorithms does not affect the simpler ones [202, 89, 156].

8.4.1 XML Datasets and Queries

Five datasets are used in the experiments, and their characteristics are shown in Section 6.5.1.1 of Chapter 6. Therefore, the DBLP (see Section 5.4.1.1), XMark with the scaling factor equal to 1 (see Section 5.4.2.1), TreeBank (see Section 5.4.1.2), Random (see Section 5.4.3.1) and Zipf 5.4.3.2) datasets were used in the experiments. Queries for DBLP, XMark, TreeBank and Random datasets are similar to that used in the experiments of Chapter 6. Because bottom-approaches can process recursive queries efficiently, one recursive TPQ was added to TPQs over the TreeBank dataset. Table 8.1 presents an overview of the TPQs

Table 8.1: Experimental TPQs for TreeBank.

Code	XPath expression	Result size
TQ_1	//S[//MD]//ADJ	19
TQ_2	//S/VP/PP[/NP/VBN]/IN	152
TQ_3	//VP[/DT]//PRP_DOLLAR_	3
TQ_4	//S[/JJ]/NP	5
TQ_5	//S[VP[DT]//NN]/NP	32
TQ_6	//S[//VP/IN]//NP	20311
TQ_7	//S/VP/PP[//NP/VBN]/IN	320
TQ_8	//EMPTY/S//NP[/SBAR/WHNP/PP//NN]/_COMMA_	17
TQ_9	//SINV//NP[/PP//JJR][//S]/NN	4
TQ_{10}	//NP[/NN]/PP	43942
TQ_{11}	//S[//VP][//NP]/VP/PP[IN]/NP/VBN	1185

Table 8.2: Zipf TPQ templates for XPath expressions.

Template	Query template	# of randomly generated queries
T_1	$//\alpha/\beta[//\chi]/\delta$	10
T_2	$//\varepsilon[/\eta]/\gamma$	10
T_3	$//\alpha[/\varepsilon][/\eta]//\gamma$	10
T_4	$//\alpha/\beta//\chi/\delta$	10
T_5	$//\alpha/\alpha[//\beta]/\chi//\chi[/\delta]//\varepsilon$	10

over the TreeBank dataset. The XML structured queries for evaluation over the Zipf dataset were generated according to five query templates shown in Table 8.2. Templates specify relationships between query nodes. For each template, ten TPQs were randomly generated such that $\alpha,\beta,\chi,\delta \in \{a,b,d,g\}$ and $\varepsilon,\eta,\gamma \in \{a,b,c,d,e,f,g\}$. A list of the generated Zipf queries can be found in Appendix C. In order to show the difference between the tested algorithms and make the comparisons more comprehensive, ten recursive TPQs were generated according to the template T_5 (see Table 8.2).

8.4.2 Metrics

The experiments compare two variables for each TPQ selected under the bottom-up holistic algorithms. Accordingly, the performance comparison of these algorithms was based on the following metrics:

- Number of elements stored in the intermediate storage: the number of elements stored by each algorithm in the intermediate storage.

- Processing time: the running time of an algorithm spent on the whole TPQ includes the filtering process and the result enumeration (in milliseconds). All TPQs were executed 103 times to increase the reliability of measures and the first three runs were excluded for cold cache issues. The I/O cost for tag indexing files for the set of

algorithms in "CPL" and "CPL_" is not counted because it is negligible, and the cost to read the tag indexing is constant over a series of TPQs for each dataset. By way of explanation, the tag indexing needs only to be read once for a set of TPQs over a particular dataset.

8.4.3 Experimental Results

This section presents the evaluation of the experimental results. In order to verify the validity of the new approaches, all query results in the experiments returned from the tested algorithms were inspected. Since all algorithms returned the same results, the validity, can thus be verified. To allow precise comparisons, the discussion of the query performance related to a particular dataset is contained within an individual subsection. The query performances for TPQs over DBLP, XMark, TreeBank, Random and Zipf datasets are discussed in Sections 8.4.3.1, 8.4.3.2, 8.4.3.3, 8.4.3.4 and 8.4.3.5, respectively. In addition, for each dataset, the experimental results of processing time are depicted in two graphs based on the intermediate storage approach used, the reason for this is to enable a clear presentation of several algorithms in the same plot. To avoid extreme value differences, experimental results of TwigList regarding the number of elements stored in the intermediate storage are not included in the illustrative graphs since the remaining algorithms use an improved version of TwigList. A complete result including TwigList is presented in Appendix C. The scalability tests are evaluated in Section 8.4.3.6.

8.4.3.1 DBLP

This section discusses the experimental results for TPQs over the DBLP dataset, the TPQs are given in Table 6.5. As it is not common for TPQs, which contain both '//' and '/', to have a significant difference in performance for data-oriented datasets, the purpose of this dataset is to show that the new approaches do not bring any overhead for processing XML documents with a simple structure. Figure 8.10 shows the number of elements stored in the intermediate storage by each algorithm along with the actual elements which are parts of complete matches. An immediate observation from the figure is that "CPL" and "CPL_" approaches are more efficient in terms of the intermediate results than the tested algorithms for all queries on this dataset. Even though the DBLP document is highly-structured, the state-of-the-art TJStrictPre, TJStrictPost and GTPStack algorithms provided optimal evaluation only for one TPQ, namely DQ_2. Moreover, the size of the intermediate storage for TwigList and twigFast was 580 and 2 times larger respectively than that constructed by the new approaches. The rest of the algorithms tested appended useless elements accounted for 30% to 40% of all relevant elements.

Moving on now to compare query processing time for this dataset, the performance of the algorithms using the simple list approach is illustrated in Figure 8.11a while Figure 8.11b shows the experimental results for algorithms utilising the level split approach.

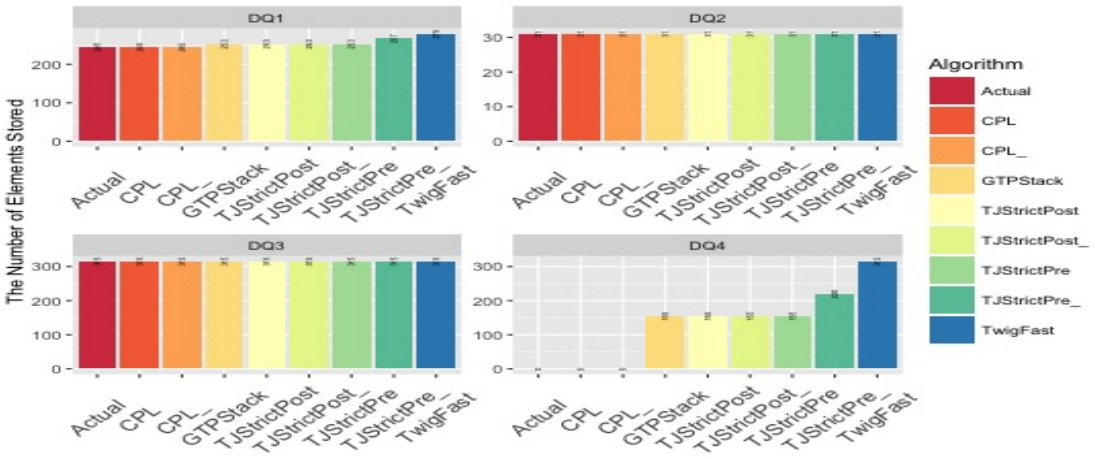


Figure 8.10: The number of elements stored in the intermediate storage by each algorithm for the queries tested over DBLP. "Actual" represents the number of elements relevant to the query results.

Table 8.3: Results for the comparison groups over the DBLP document.

Query	p-value	p-value < 0.05
DQ1	≈ 0	TRUE
DQ2	≈ 0	TRUE
DQ3	$4.17e^{-181}$	TRUE
DQ4	≈ 0	TRUE

As can be seen from the experimental results, "CPL" and "CPL_" approaches outperform the comparable algorithms for all TPQs except for DQ_1, see Figures 8.11a and 8.11b. A possible explanation is that the CPL approach improves the filtering process without any additional overhead [12]. To compare statistically the overall query performance, the Kruskal-Wallis test for each TPQ was carried out to see whether there was a difference in the performance between at least two algorithms or not. Table 8.3 provides the results of running the Kruskal-Wallis tests over the queries using the significance level *0.05* to test the null hypothesis. Accordingly, it can be seen from the table that the Kruskal-Wallis tests suggest that there is a significant difference in the performance between two algorithms at least. In some queries the p-value for running Kruskal-Wallis tests is nearly zero which strongly indicates that the observed sample is improbable under the null hypothesis [1] [82, 74]. Therefore, the Mann-Whitney U test was run for all possible combinations of pair over the queries tested. This set was computed using Formula 6.2 as follows: $= \frac{(21\times(21-1))}{2} \times 4 \ = 840$. Raw timings are given in Appendix C.

The summary of the paired comparisons based on the Mann Whitney U test is provided in Table 8.4. The table contains the number of comparisons for which a particular algorithm was faster and slower. The tested algorithms are sorted according to the performance speed in descending order. The reason for this is that the number of comparisons for which

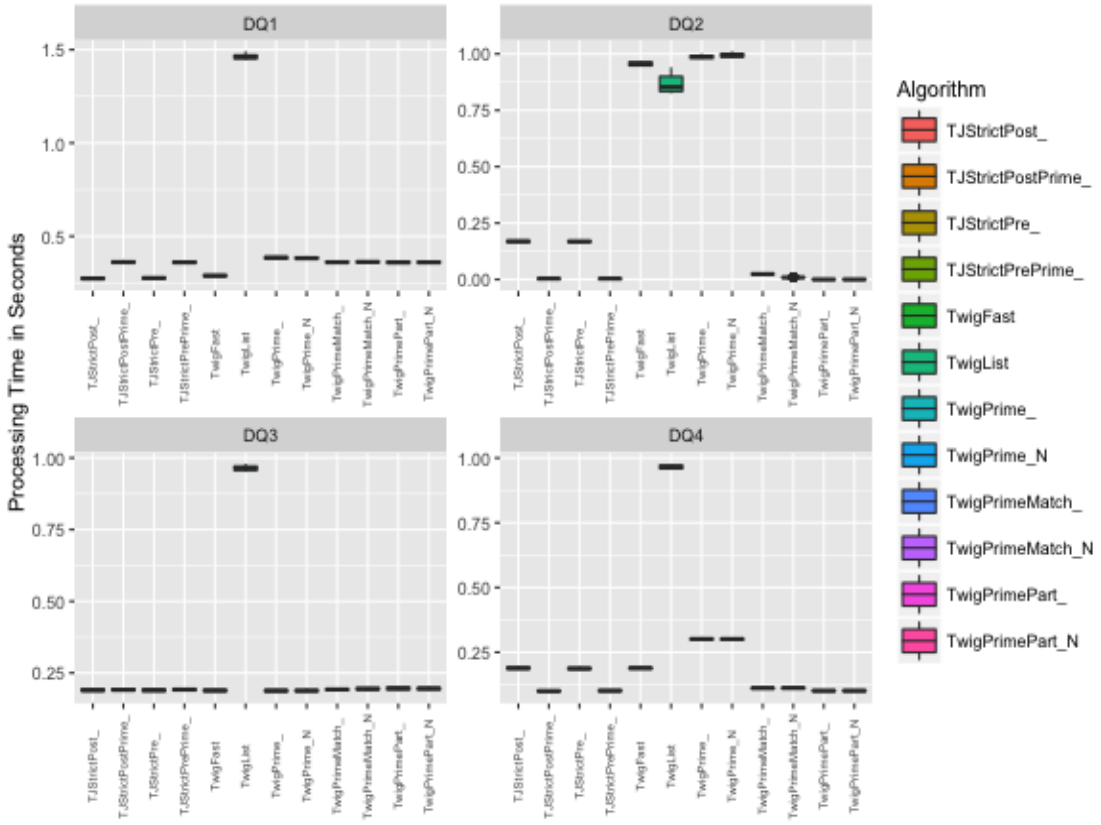
[1] These p-values don't indicate that the p-value is literally zero. They are simply less than the smallest representable positive double-precision floating point value in R [182].

an algorithm was faster is misleading because most of the algorithms in the experiments belong to "CPL" and "CPL_" groups. For instance, the TJStrictPrePrime_ showed similar performance with the algorithms compared in 19 cases while it was slower in 21 cases. On the other hand, the GTPStack algorithm has a better performance in 56 comparisons but it was slower in more cases than TJStrictPostPrime_. Moreover, TJStrictPostPrime_ performed better than GTPStack in two queries whereas GTPStack outperformed TJStrict-PostPrime_ in two cases. As shown in Table 8.4, the CPL approaches outperformed the existing algorithms. TwigPrimePart_N which can be seen as a combination of TwigPrime and *getPart* outperformed both approaches TJStrictPre and TJStrictPost using the *getPart*. However, TwigPrime using the *getNext* performed very poorly for this dataset because the *getPart* and *getMatch* can skip irrelevant elements efficiently using the cursor forward movement.

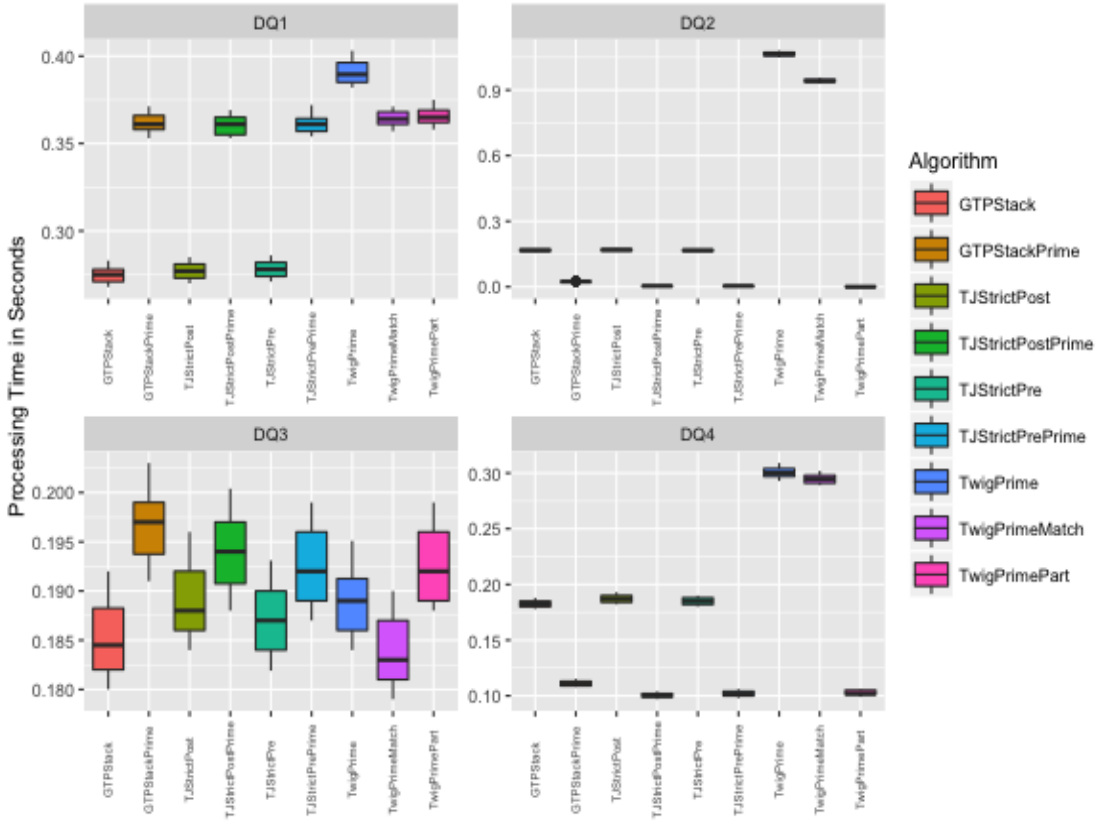
Table 8.4: The overall comparisons based on U tests over the DBLP dataset. "-" indicates no difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
TJStrictPostPrime_	45	18	17
GTPStack	56	20	4
TJStrictPrePrime	40	21	19
TJStrictPrePrime_	40	21	19
TJStrictPostPrime	43	22	15
TwigPrimePart_N	40	24	16
TwigPrimePart_	40	25	15
TJStrictPre	50	26	4
TJStrictPost	40	29	11
TJStrictPost_	40	29	11
TwigPrimePart	41	30	9
TJStrictPre_	40	30	10
TwigPrimeMatch_	34	31	15
TwigPrimeMatch_N	33	35	12
GTPStackPrime	29	38	13
TwigFast	35	39	6
TwigPrimeMatch	31	47	2
TwigPrime_	18	54	8
TwigPrime_N	17	55	8
TwigPrime	13	59	8
TwigList	5	75	0

To conclude, The "CPL" and "CPL_" approaches stored only elements contributing to the query results in the experiments. Except TwigList, the rest of the tested approaches stored useless elements up to 60% more than the relevant elements. The CPL approaches showed a better performance in all cases, however, the existing algorithms were comparable to the new approaches only for DQ_1. This can be explained because few elements in the dataset violate the structural relationships specified by DQ_1.



(a)



(b)

Figure 8.11: Query processing time of the algorithms utilising the simple list approach in (a) and the level split approach in (b) against the DBLP dataset.

8.4.3.2 XMark

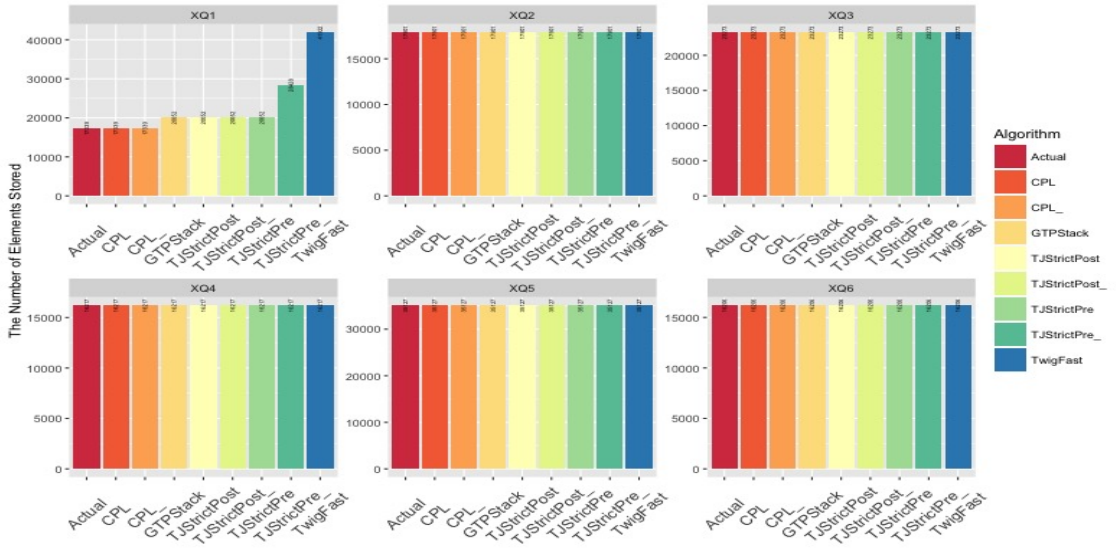


Figure 8.12: The number of elements stored in the intermediate storage by each algorithm for the queries tested over XMark. "Actual" represents the number of elements relevant to the query results.

In the XMark dataset, the experiment is to compare the performance of the algorithms on a relatively balanced XML tree. The number of elements stored by each algorithm are shown in Figure 8.12. Similar to the experimental results obtained from DBLP, both "CPL" and "CPL_" approaches showed optimal performance, whereas the existing algorithms using the *getNext* can not perform efficiently for XQ_1. On the other hand, TwigList showed the worst performance because it uses a simple filtering strategy which skips elements according to parent query nodes and does not perform cursor forward movement. Figure 8.13 presents query processing overall performance for this experiment.

To compare the query performance, the experiment is based on hypothesis that there is no difference in the performance between the algorithms so that the Kruskal-Wallis test was carried out to test that null hypothesis. Table 8.5 provides an overview of group-based comparisons using Kruskal-Wallis tests. There is a significant difference in the performance between two algorithms at least for all TPQs as suggested by Kruskal-Wallis tests [2], thus the total number of paired comparisons using Formula 6.2 is computed as follows: $= \frac{(21\times(21-1))}{2} \times 6 = 1260$. The raw data of query processing time for the XMark dataset can be found in Appendix C.

Combining the figures from Table 8.6 and the experimental results given in Figures 8.13a and 8.13b, TJStrictPost_ and TJStrictPre_ using the simple list approach showed the best performance in most comparisons. These results could be due to the benefits of using the level split approach seem to be exceeded by the cost of maintaining and accessing them.

[2] In all queries the p-value for running Kruskal-Wallis tests is nearly zero which strongly indicates that the observed sample is improbable under the null hypothesis. These p-values don't indicate that the p-value is literally zero. They are simply less than the smallest representable positive double-precision floating point value in R [182].

Table 8.5: Results for the comparison groups over XMark dataset.

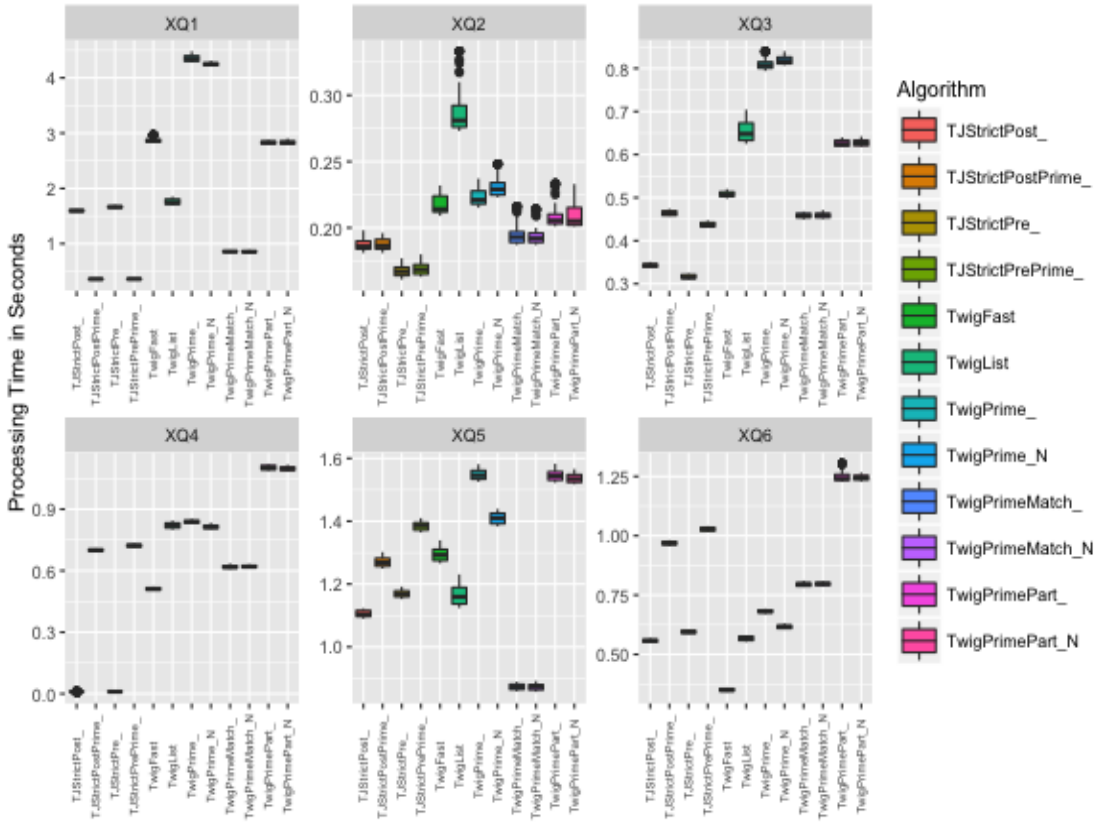
Query	p-value	p-value < 0.05
XQ1	≈ 0	TRUE
XQ2	≈ 0	TRUE
XQ3	≈ 0	TRUE
XQ4	≈ 0	TRUE
XQ5	≈ 0	TRUE
XQ6	≈ 0	TRUE

TwigPrimeMatch_ and TwigPrimeMatch_N showed a better performance compared to the rest of the algorithms tested. Inefficiency of the GTPStack and GTPStackPrime comes from the use of linked lists as the intermediate storage which increases the cost of enumerating results since linked list do not support random access efficiently. They were significantly worse than the other algorithms in three queries. The raw data for all paired comparisons is presented in Appendix C. The CPL approaches were comparable to TJStrictPost_ and TJStrictPre_ in all queries except XQ_4, and the new algorithm TwigPrime combined with the *getMatch* function outperformed them in two queries.

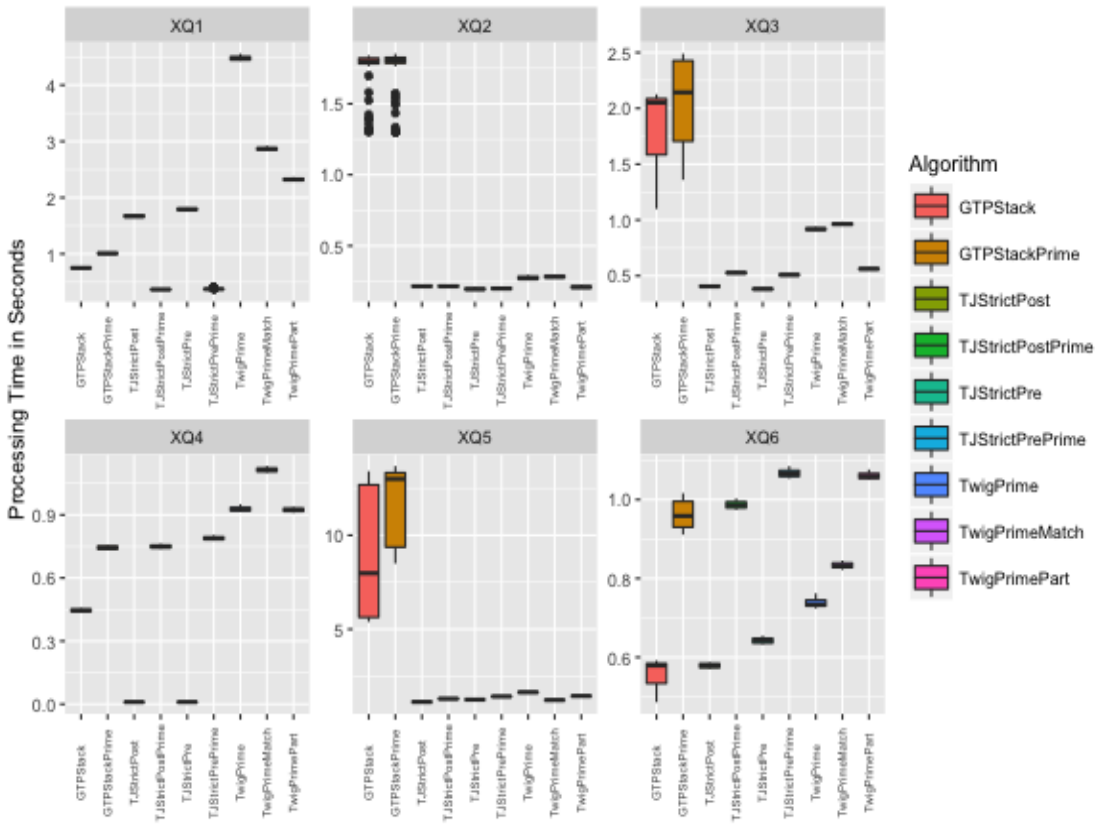
Table 8.6: The overall comparisons based on U tests for all queries in the XMark dataset. "-" indicates no difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
TJStrictPost_	102	11	7
TJStrictPre_	97	19	4
TJStrictPost	84	29	7
TwigPrimeMatch_	84	30	6
TwigPrimeMatch_N	84	30	6
TJStrictPre	83	34	3
TJStrictPostPrime_	79	39	2
TJStrictPrePrime_	78	41	1
TwigFast	67	49	4
TJStrictPostPrime	59	59	2
TJStrictPrePrime	58	61	1
TwigList	57	62	1
GTPStack	50	68	2
TwigPrime_N	40	80	0
TwigPrimePart	39	81	0
TwigPrimePart_N	30	86	4
TwigPrime_	32	87	1
TwigPrimePart_	27	88	5
TwigPrimeMatch	29	89	2
GTPStackPrime	30	89	1
TwigPrime	23	97	0

To conclude, the experimental results demonstrated that the CPL approaches had a superior performance to the comparable algorithms in terms of the number of elements pushed into the intermediate results. For the XMark dataset, the CPL approach efficiently filters out useless elements in all queries so that the use of semi-strict subtree filtering checks followed by a strict prefix path matching can provide optimal evaluation. There is no need to perform a strict subtree filtering using the level split approach. Thus, a promising approach to process efficiently the XMark queries might involve the combination of TwigPrime and the *getMatch* function such as TwigPrimeMatch_ and TwigPrimeMatch_N. To summarise, the findings of this analysis suggest that the semi-strict filtering developed in this thesis can be considered to evaluate efficiently TPQs on data-oriented dataset. It is possible that the performance gain of TJStrictPost and TJStrictPost_ is achieved because of elements are pushed into the intermediate storage in postorder and a strict subtree filtering can be performed with additional extra passes over the intermediate results. However, these approaches output the result tuples unordered, see Definition 4.11.



(a)



(b)

Figure 8.13: Query processing time of the algorithms using the simple list approach in (a) and the level split approach in (b) against the XMark dataset.

8.4.3.3 TreeBank

Eleven TPQs including complex and recursive queries (see Table 8.1) were used in the TreeBank dataset to understand the differences among the algorithms where the XML document has a highly recursive structure. Figure 8.14 shows the size of the intermediate storage constructed by each algorithm. In comparison to the previous datasets, the TreeBank dataset is the most complicated XML document from various aspects of query processing point of view [255, 23, 147] although it is the smallest dataset in the experiments. Therefore, the suboptimal evaluation of the existing approaches can be demonstrated here as depicted in the illustrative graph. This dataset may also demonstrate the effectiveness of the CPL approaches. In contrast to the previous experiments in Sections 8.4.3.1 and 8.4.3.2, the "CPL" and "CPL_" approaches failed to provide optimal evaluation for TQ_2, TQ_5, TQ_8 and TQ_{11} in which useless elements were stored. They stored several orders of magnitude fewer elements than the comparable algorithms. For TQ_6, the "CPL" and "CPL_" approaches performed efficiently by storing only the 16,062 useful elements, whereas the number of elements stored in the state-of-the-art algorithms was between 374,370 and 563,741. The simple algorithms TwigList and TwigFast built up the intermediate storage with 770,052 and 669,312 elements in order to evaluate TQ_6, respectively.

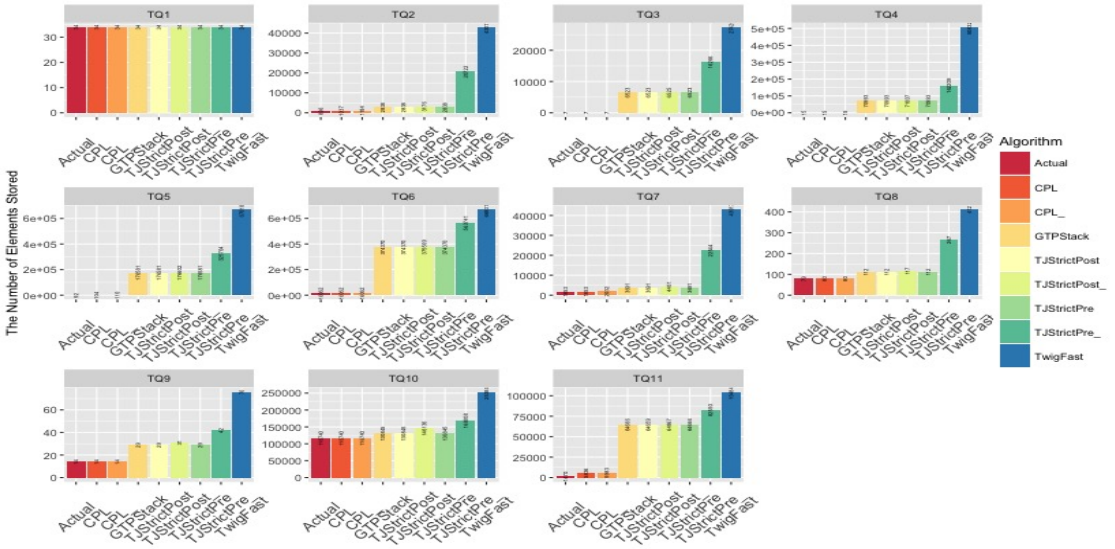


Figure 8.14: The number of elements stored in the intermediate storage by each algorithm for the queries tested over the TreeBank document. Actual represents the number of elements relevant to the query results.

Figure 8.15 shows the execution time of the algorithms over this dataset. For better presentation the algorithms tested are grouped based on the approach used to store the intermediate results. In Figure 8.15a, the query processing times for algorithms based on the simple list approach are given while Figure 8.15b presents the overall query performance for the level split approaches. In order to test the group comparisons, eleven Kruskal-Wallis tests were carried out over TPQs to see whether there was a difference in the

Table 8.7: Results for the comparison groups over TreeBank dataset.

Query	p-value	p-value < 0.05
TQ1	≈ 0	TRUE
TQ2	≈ 0	TRUE
TQ3	≈ 0	TRUE
TQ4	≈ 0	TRUE
TQ5	≈ 0	TRUE
TQ6	≈ 0	TRUE
TQ7	≈ 0	TRUE
TQ8	≈ 0	TRUE
TQ9	≈ 0	TRUE
TQ10	≈ 0	TRUE
TQ11	≈ 0	TRUE

performance between the algorithms or not. All the Kruskal-Wallis tests suggested that there is a difference in the performance between two algorithms at least as presented in Table 8.7. The overall results suggest that the sample is improbable under the null hypothesis[3]. Consequently, the paired comparisons based on the U test of Mann Whitney were calculated. The number of paired comparisons for this dataset can be obtained using Formula 6.2 as $= \frac{(21\times(21-1))}{2} \times 11 \ = 2310$. The raw data of query processing time for the real-world dataset, TreeBank can be found in Appendix C.

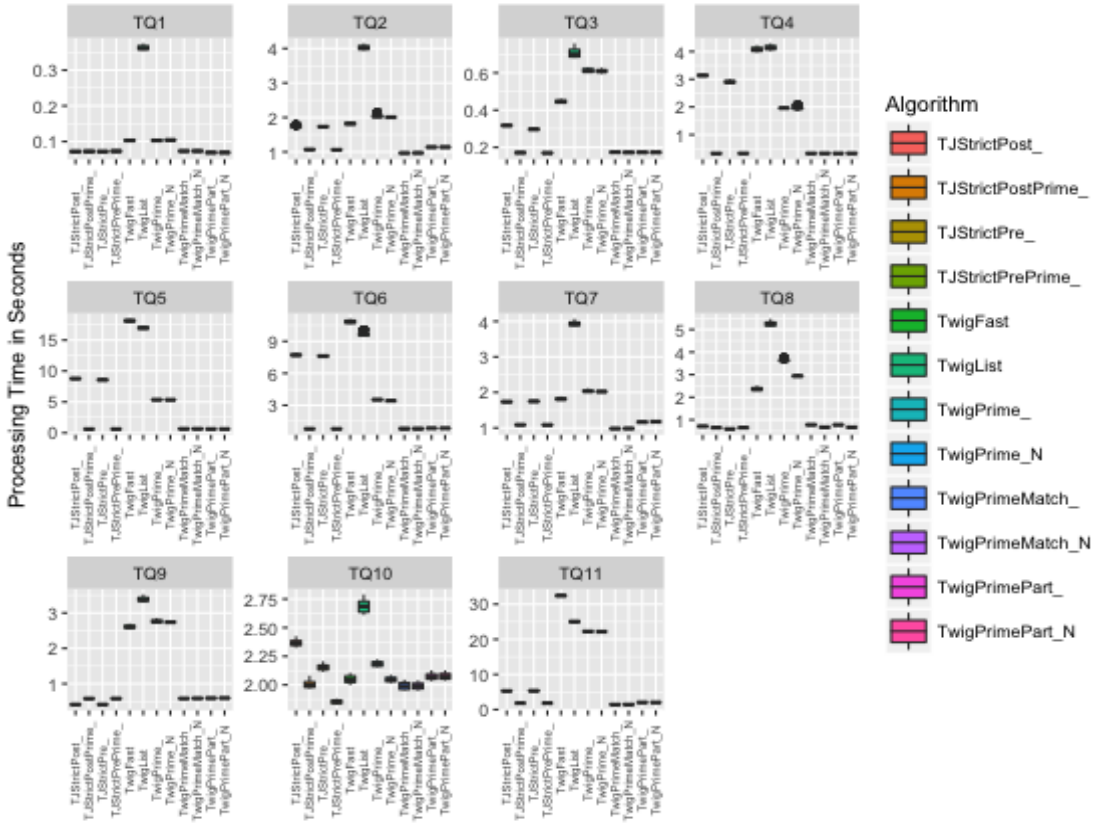
The summary of the paired comparisons based on the Mann Whitney U test is presented in Table 8.8. It can be seen from the data in Table 8.8 and Figures 8.15a and 8.15b that the "CPL" and "CPL_" approaches significantly outperformed the rest of the algorithms tested. Eleven different versions of the new approaches significantly outperformed the other algorithms, and the combination of TwigPrime and the *getMatch* function using the simple list approach showed a superior performance to the other combinations of TwigPrime. The reason for this is due to the use of the CPL approach to filter useless elements and the *getMatch* to avoid redundant computations. When the new algorithm TwigPrime uses the level split approach, the *getPart* function had the best performance. This is due to the use of additional vector which stores one extra value for each query node to check the latest ancestors that form a weak full match for the entire TPQ in *getPart* while *getMatch* has to check several tails and ancestors to determine whether an element is useful or not. Note that TQ_6 is a very expensive query in the experiments, it touches a very large portion of the document and the answer to it is a quite large. It was chosen because it shows the effects of the CPL approach. For TQ_6, only the "CPL" and "CPL_" algorithms can provide optimal evaluation and hence the reduction in the CPU cost of the algorithms.

[3]In all queries the p-value for running Kruskal-Wallis tests is nearly zero which strongly indicates that the observed sample is improbable under the null hypothesis. These p-values don't indicate that the p-value is literally zero. They are simply less than the smallest representable positive double-precision floating point value in R [182]

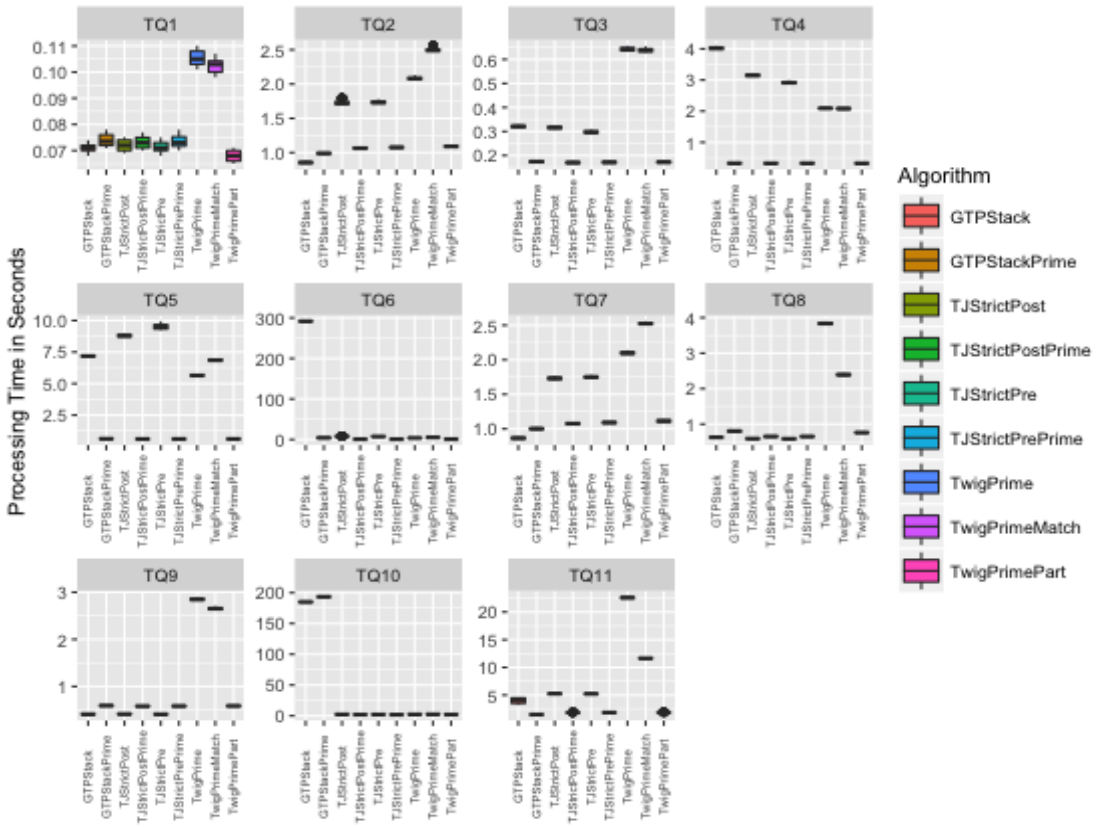
Table 8.8: The overall comparisons based on U tests for all queries in the TreeBank dataset. "-" indicates no difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
TwigPrimeMatch_	143	45	32
TwigPrimeMatch_N	146	47	27
TwigPrimePart	140	64	16
TwigPrimePart_N	127	75	18
TwigPrimePart_	123	77	20
TJStrictPrePrime	88	119	13
TJStrictPrePrime_	92	119	9
TJStrictPostPrime	92	121	7
TJStrictPostPrime_	96	121	3
TwigPrime_N	65	149	6
GTPStackPrime	59	152	9
GTPStack	61	159	0
TJStrictPre	61	159	0
TwigPrime_	53	161	6
TJStrictPre_	56	164	0
TJStrictPost	55	165	0
TJStrictPost_	52	168	0
TwigFast	42	174	4
TwigPrime	39	181	0
TwigPrimeMatch	28	190	2
TwigList	6	214	0

To sum up, the "CPL" and "CPL_" approaches showed a superior performance to the other algorithms in terms of the number of elements stored and query running time for a highly recursive dataset. For TPQs falling in the optimal class of the CPL approach, the new algorithms did not store useless elements, whereas the others did. This verifies the analysis of its optimal sets of TPQs as stated in Section 8.3.1.1.



(a)



(b)

Figure 8.15: Query processing time of the algorithms using the simple list approach in (a) and the level split approach in (b) against the TreeBank dataset.

8.4.3.4 Random

The Random dataset has a complex structure with six distinct tags. This dataset was selected to show the differences between algorithms where the XML combines features of DBLP and TreeBank, being relatively structured and deeply recursive at the same time. The size of intermediate storage generated by each algorithm is presented in Figure 8.16

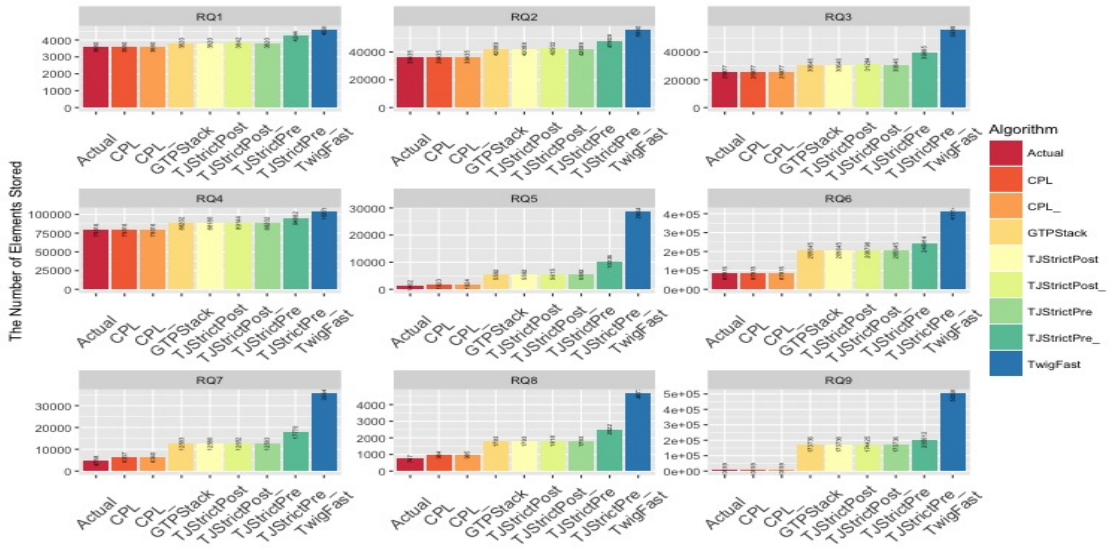


Figure 8.16: The number of elements stored in the intermediate storage by each algorithm for the queries tested over the Random dataset. "Actual" represents the number of elements relevant to the query results.

Six TPQs, namely RQ_1, RQ_2, RQ_3, RQ_4, RQ_6 and RQ_9, can be processed efficiently by the CPL approaches as illustrated in Figure 8.16. For the remaining queries, the CPL algorithms stored by far fewer elements than the other methods tested. Both RQ_6 and RQ_9 show the effects of introducing the CPL approach to the existing advanced preorder filtering functions. It can be observed that even for queries which do not fall within the optimal sets of TPQs for the CPL approach, optimal evaluation can be achieved as in the case of RQ_6, and the new approaches significantly outperformed the other algorithms. For instance, in RQ_9, the number of elements stored in the "CPL" and "CPL_" approaches was 17 times less than that stored in the existing algorithms using the level split intermediate storage, and up to three orders of magnitude fewer than methods using simple vectors. Clearly, the size of intermediate storage built by the new approach is by far less than the other approaches.

To compare the query performance, Kruskal-Wallis test was carried out to test the null hypothesis for each TPQ tested. The result of group comparisons based on Kruskal-Wallis test are summarised in Table 8.9. Since the results turned out to reject the null hypothesis by suggesting that for each TPQ there was a significantly difference in the performance between two algorithms at least[4], all the possible paired comparisons were computed using

[4]In all queries the p-value for running Kruskal-Wallis tests is nearly zero which strongly indicates that the observed sample is improbable under the null hypothesis. These p-values don't indicate that the p-value

Table 8.9: Results for the comparison groups over the Random dataset.

Query	p-value	p-value < 0.05
RQ1	≈ 0	TRUE
RQ2	≈ 0	TRUE
RQ3	≈ 0	TRUE
RQ4	≈ 0	TRUE
RQ5	≈ 0	TRUE
RQ6	≈ 0	TRUE
RQ7	≈ 0	TRUE
RQ8	≈ 0	TRUE
RQ9	≈ 0	TRUE

Formula 6.2 as follows: $= \frac{(21\times(21-1))}{2} \times 9 \ = 1890$. The full results are given in Appendix C.

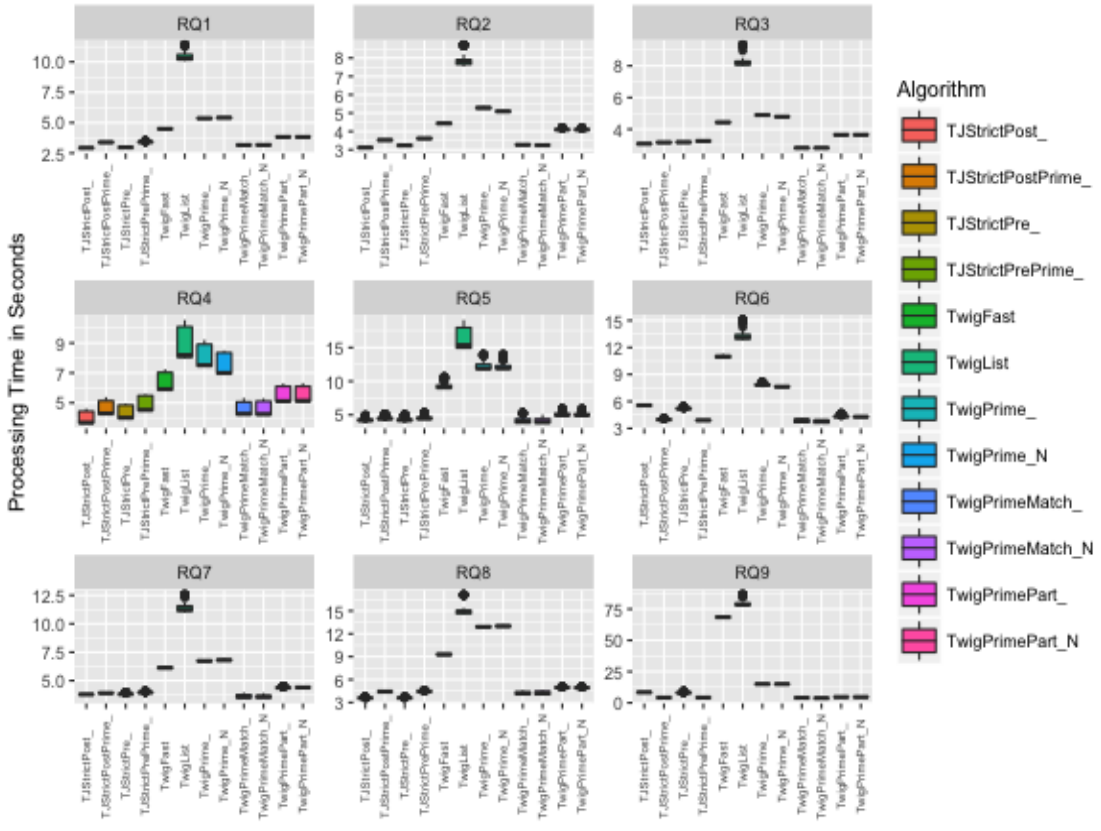
The results obtained from all paired comparisons based on the Mann Whitney U test of query running time are summarised in Table 8.10. It is apparent from this table that when TwigPrime is combined with the advanced preorder filtering strategy, i.e., *getMatch*, the "CPL_" approach significantly performed better than the "CPL" approaches. Only TwigPrimePart which uses the *getPart* showed a superior performance to the other versions of TwigPrime on the level split approach (i.e., the "CPL" approaches). Interestingly, TwigPrimePart performed better than the other approaches using combinations of the *getPart* function and the CPL approach, such as TJStrictPrePrime and TJStrcitPrePrime_. The enhanced performance can illustrate that maintaining pointers to perform a strict prefix path matching may provide low overhead when compared to stack operations for highly recursive datasets including the TreeBank and Random dataset. Moreover, when TPQs contain many P-C edges, the "CPL_" and "CPL" approaches performed better than the existing algorithms, which can be observed in RQ_9 (see Figures 8.17a and 8.17b).

In summary, the "CPL_" approaches can be considered when evaluating TPQs in this type of XML dataset. They provide optimal evaluation in several queries and get better performance for complex queries. This is because the CPL approach can filter out many useless elements before storing them in the intermediate storage while the *getMatch* function avoids unnecessary recursive calls during the query processing. Therefore, the overall CPU cost can be saved. The benefits of using level split intermediate results combined with the CPL approach seems to be outweighed by the cost of maintaining and accessing them since the combined filtering can prevent slightly fewer elements than that stored using the CPL approach alone (see Figure 8.16).

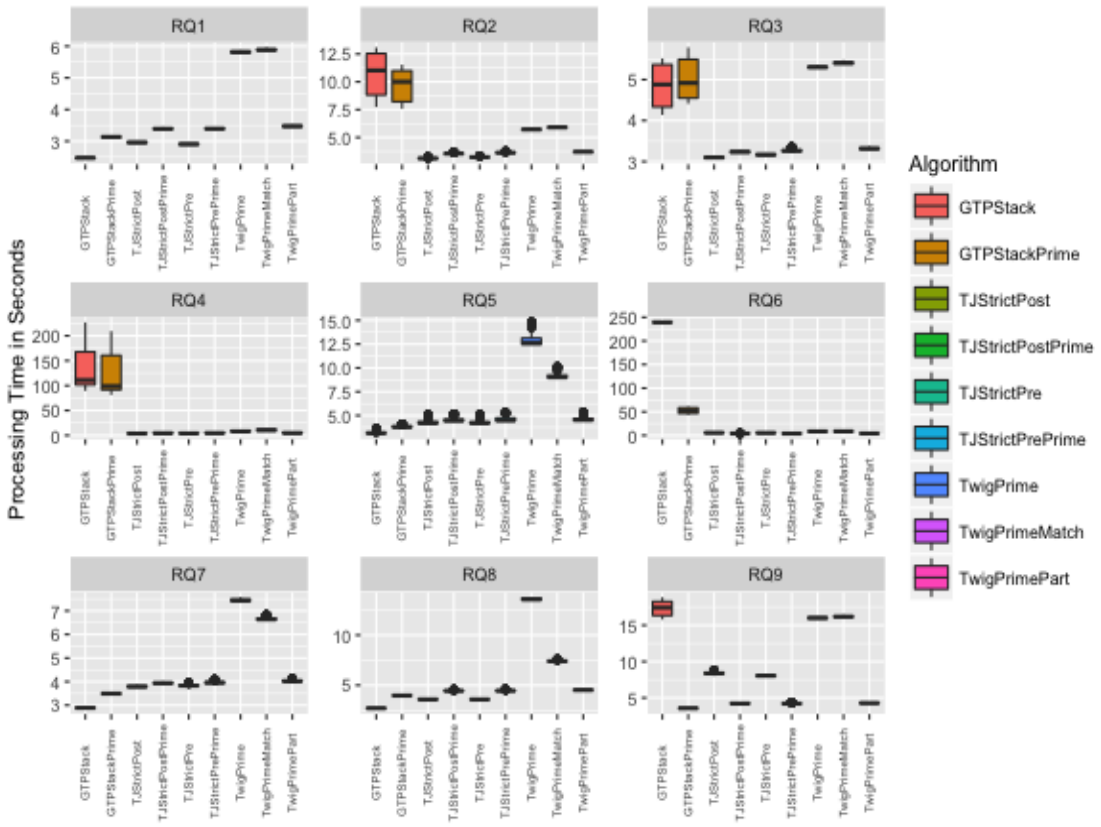
is literally zero. They are simply less than the smallest representable positive double-precision floating point value in R [182]

Table 8.10: The overall comparisons based on U tests for all queries in the Random dataset. "-" indicates no difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
TwigPrimeMatch_N	149	25	6
TwigPrimeMatch_	145	28	7
TJStrictPre	133	40	7
TJStrictPre_	125	50	5
TJStrictPost_	93	83	4
TJStrictPost	92	84	4
TwigPrimePart	89	91	0
TJStrictPostPrime_	86	93	1
TJStrictPrePrime	85	95	0
TJStrictPrePrime_	85	94	1
TJStrictPostPrime	85	95	0
TwigPrimePart_N	73	101	6
TwigPrimePart_	72	102	6
GTPStackPrime	62	116	2
GTPStack	57	121	2
TwigFast	43	137	0
TwigPrime_N	37	139	4
TwigPrime_	36	140	4
TwigPrimeMatch	28	152	0
TwigPrime	24	156	0
TwigList	7	173	0



(a)



(b)

Figure 8.17: Query processing time of the algorithms using the simple list approach in (a) and the level split approach in (b) against the Random dataset.

8.4.3.5 Zipf

The Zipf collection and queries were used to gain insight into the benefits and the potential overheads of using combinations of the CPL approach, different advanced preorder filtering functions (i.e., *getNext*, *getPart* and *getMatch*) and the level split intermediate results. As a result, only algorithms using the level split approach to build the intermediate storage and storing elements in preorder were compared. In addition, approaches which store elements in the intermediate storage in postorder, such as TJStrictPost and TJStrictPostPrime, were not included in the performance comparison because they output the result tuples unordered, hence they are not directly comparable. Recall that every element in the Zipf dataset has exactly two children and the longest path in the document is 26 (see Section 5.4.3.2). Fifty queries were randomly generated according to five query templates provided in Table 8.2. For the purpose of analysis, ten queries were produced for each query template. By way of illustration, ZQ12 refers to the second Zipf query generated by template T_2. The Zipf queries generated by templates T_2, T_3 *and* T_4 can be processed efficiently by the CPL approach. On the other hand, the Zipf queries produced by template T_1 and T_5 fall outside the optimal groups of the CPL approach as stated in the analysis in Section 8.3.1.1. Note that the Zipf queries generated by template T_4 are path (i.e., non-branching) queries. The path queries were used to explore the benefits of the CPL approach although the existing algorithms can provide optimal evaluation for simple path queries (see Figure 8.19a). The size of intermediate storage generated by each algorithm is presented in Figures 8.18 and 8.19. Experimental results related to the Zipf queries generated by each template are depicted in an individual plot.

Clearly, the CPL approaches provided optimal evaluation for the Zipf queries generated by templates T_1, T_2, $T3$ *and* T_4, while the existing approaches stored useless elements as eight times as larger than the relevant elements. All the algorithms tested provided optimal evaluation for queries generated by T_4. Furthermore, the Zipf queries produced by template T_5 are recursive, complex TPQs which were chosen to show the potential effects of the distribution of P-C relationships and duplicate query nodes in TPQs. For queries, ZQ_{41} to ZQ_{50}, the CPL algorithms stored order of magnitude fewer elements than the rest of the approaches due to the fact that it uses the combined approach. The size of the intermediate storage produced by the CPL approaches for queries generated by T_5 was about 0.0134% of the size of the intermediate storage produced by the existing algorithms.

Figures 8.20 and 8.21 show query processing overall performance for this experiment. To evaluate the query performance, the Kruskal-Wallis test was carried out to test the null hypothesis stating that there is no difference in the performance between the compared algorithms. The results of the groups analysis are presented in Table 8.11. From the data in Table 8.11, every Kruskal-Wallis test revealed that there is a significant difference between two algorithms at least.

Accordingly, the total number of paired comparisons for the Zipf dataset can be computed using Formula 6.2 described in Chapter 6 as $= \frac{(7\times(7-1))}{2} \times 50 \ = 1050$. Raw timings

Table 8.11: Results for the comparison groups over the Zipf dataset.

Query	P-value	< 0.05	Query	p-value	< 0.05	Query	p-value	< 0.05
ZQ1	6.84E-136	TRUE	ZQ21	6.84E-136	TRUE	ZQ41	6.84E-136	TRUE
ZQ2	4.49E-144	TRUE	ZQ22	4.49E-144	TRUE	ZQ42	4.49E-144	TRUE
ZQ3	1.01E-141	TRUE	ZQ23	1.01E-141	TRUE	ZQ43	1.01E-141	TRUE
ZQ4	2.32E-144	TRUE	ZQ24	2.32E-144	TRUE	ZQ44	2.32E-144	TRUE
ZQ5	9.26E-143	TRUE	ZQ25	9.26E-143	TRUE	ZQ45	9.26E-143	TRUE
ZQ6	5.89E-142	TRUE	ZQ26	5.89E-142	TRUE	ZQ46	5.89E-142	TRUE
ZQ7	3.96E-142	TRUE	ZQ27	3.96E-142	TRUE	ZQ47	3.96E-142	TRUE
ZQ8	1.52E-144	TRUE	ZQ28	1.52E-144	TRUE	ZQ48	1.52E-144	TRUE
ZQ9	2.31E-144	TRUE	ZQ29	2.31E-144	TRUE	ZQ49	2.31E-144	TRUE
ZQ10	3.63E-142	TRUE	ZQ30	3.63E-142	TRUE	ZQ50	3.63E-142	TRUE
ZQ11	6.84E-136	TRUE	ZQ31	6.84E-136	TRUE			
ZQ12	4.49E-144	TRUE	ZQ32	4.49E-144	TRUE			
ZQ13	1.01E-141	TRUE	ZQ33	1.01E-141	TRUE			
ZQ14	2.32E-144	TRUE	ZQ34	2.32E-144	TRUE			
ZQ15	9.26E-143	TRUE	ZQ35	9.26E-143	TRUE			
ZQ16	5.89E-142	TRUE	ZQ36	5.89E-142	TRUE			
ZQ17	3.96E-142	TRUE	ZQ37	3.96E-142	TRUE			
ZQ18	1.52E-144	TRUE	ZQ38	1.52E-144	TRUE			
ZQ19	2.31E-144	TRUE	ZQ39	2.31E-144	TRUE			
ZQ20	3.63E-142	TRUE	ZQ40	3.63E-142	TRUE			

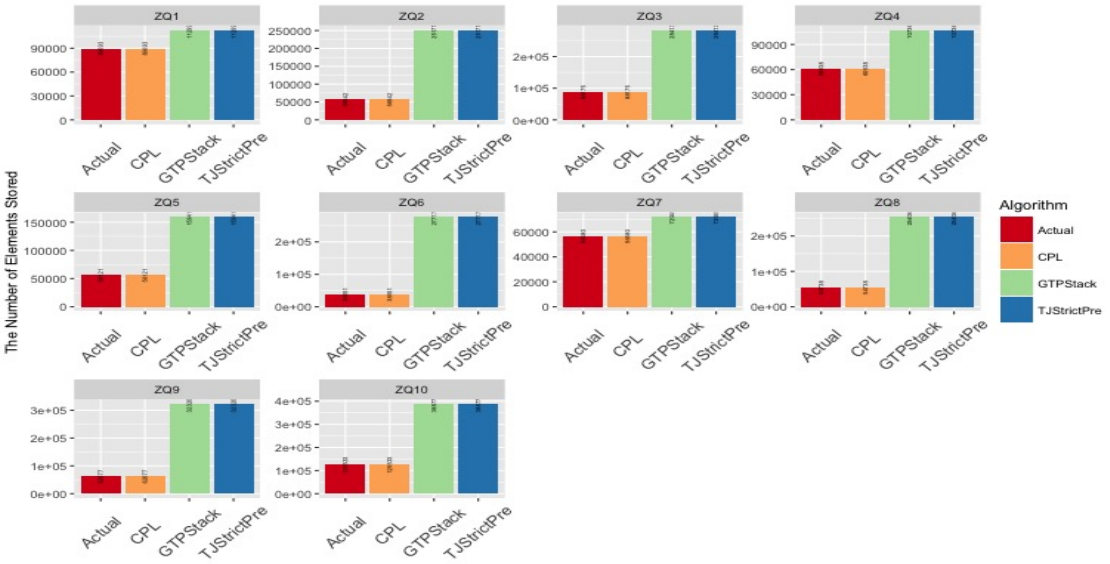
are given in Appendix C. The overall results are provided in Table 8.12 which summarises the comparisons to show how many times each algorithm statistically was either faster or slower. As depicted in illustrative graphs, the new approaches were significantly faster than the existing algorithms in most queries. TJStrictPrePrime had the best performance in most cases as it was faster in 269 out of 300 individual comparisons; however it was slower in all queries generated by template T_5 when compared with GTPStack and GTPStackPrime. The existing algorithm GTPStack was only comparable to the CPL approaches for queries corresponding to the fifth template which due to the additional filtering pass over the large intermediate results. In most queries, GTPStackPrime was faster than GTPStack because it used combined approaches, and the effect size test suggested that this had a large practical significance. Interestingly, the combination of the *getPart* function and the CPL approach performed better than any other combination as the first and the second best algorithm in performance using the *getPart* function.It seems possible that these results are due to the use of a simple technique to check a weak match for the entire query combined with advanced cursor forward movement. On the other hand, processing time of TwigPrimeMatch was influenced by the new design maintaining pointers since the latest ancestors matched are required in order to perform cursor forward movement. For every U test, the effect size suggested that there is a medium to large practical significance. However, when the null hypothesis is rejected, low practical significance was evident.

To conclude, the experiment has been used to explore the benefits of the CPL relationship when evaluating the Zipf queries according to predefined query templates. The Zipf dataset was chosen to compare the performance of the algorithms by varying the selectivity

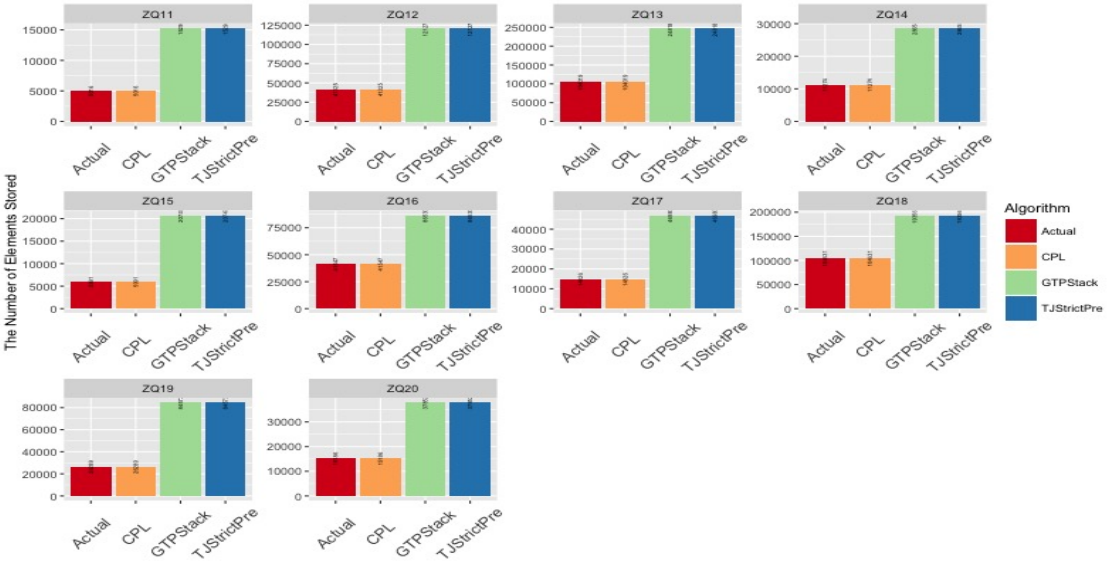
Table 8.12: The overall comparisons based on U tests for all queries in the Zipf collection. "-" indicates no difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
TJStrictPrePrime	269	29	2
TwigPrimePart	216	81	3
TwigPrime	149	148	3
TwigPrimeMatch	130	167	3
GTPStackPrime	120	175	5
TJStrictPre	106	193	1
GTPStack	48	247	5

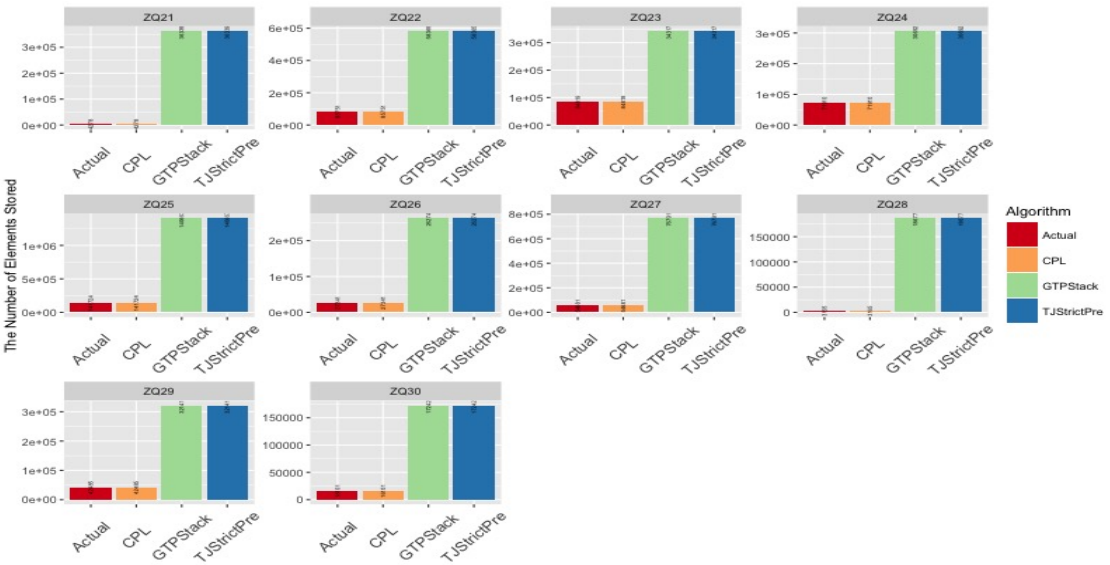
of query nodes using the property of the Zipf distribution. As a result, the CPL approaches showed superior performance to the other techniques in terms of the number of elements stored in the intermediate storage and query running time.



(a) the Zipf queries generated by template 1

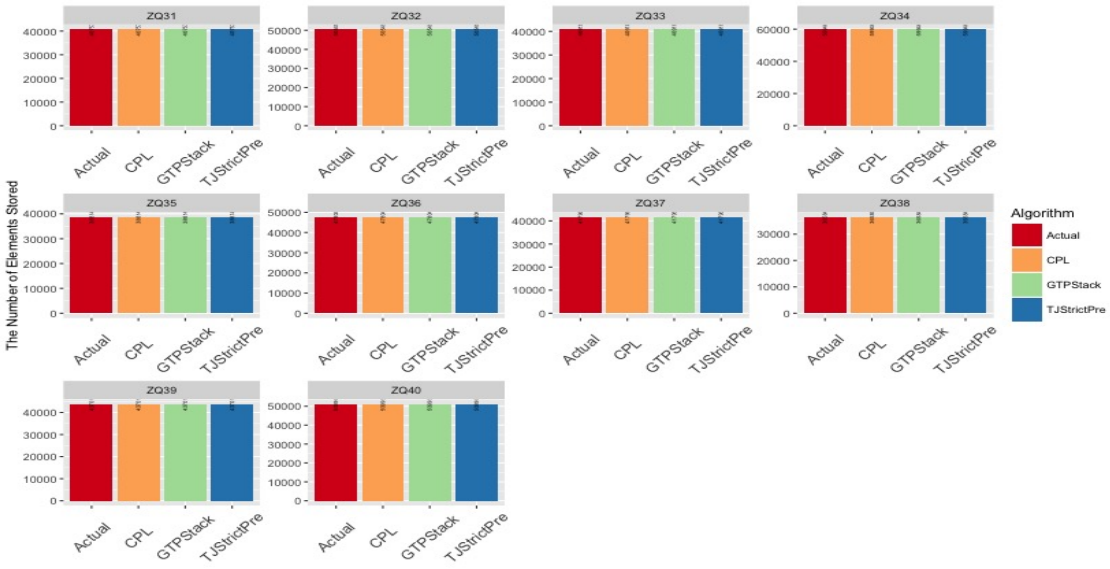


(b) the Zipf queries generated by template 2

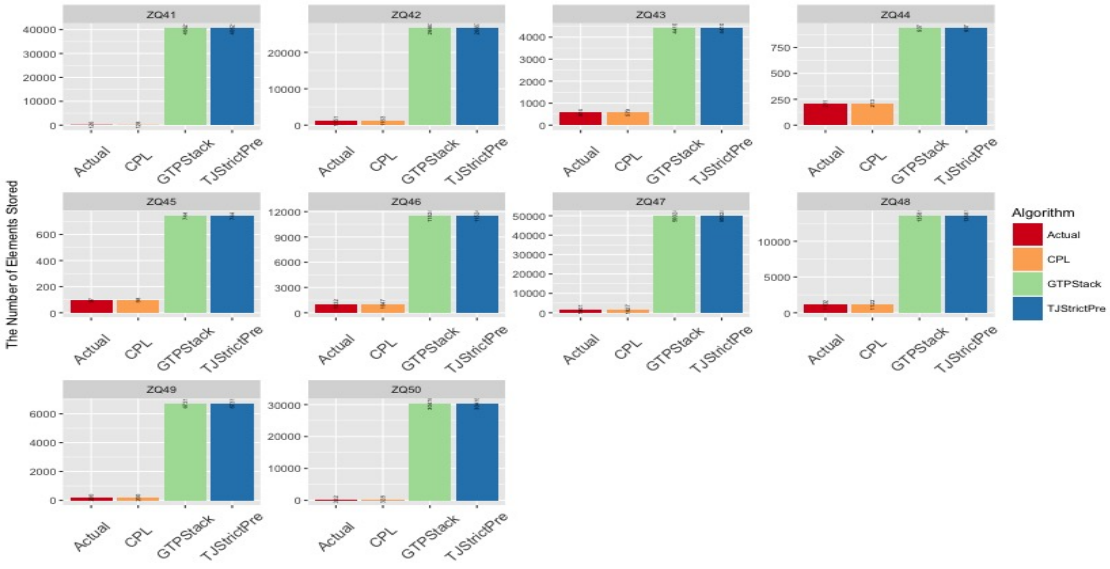


(c) the Zipf queries generated by template 3

Figure 8.18: The number of elements stored in the intermediate storage by each algorithm for the queries tested over the Zipf dataset. Actual represents the number of elements relevant to the query results.

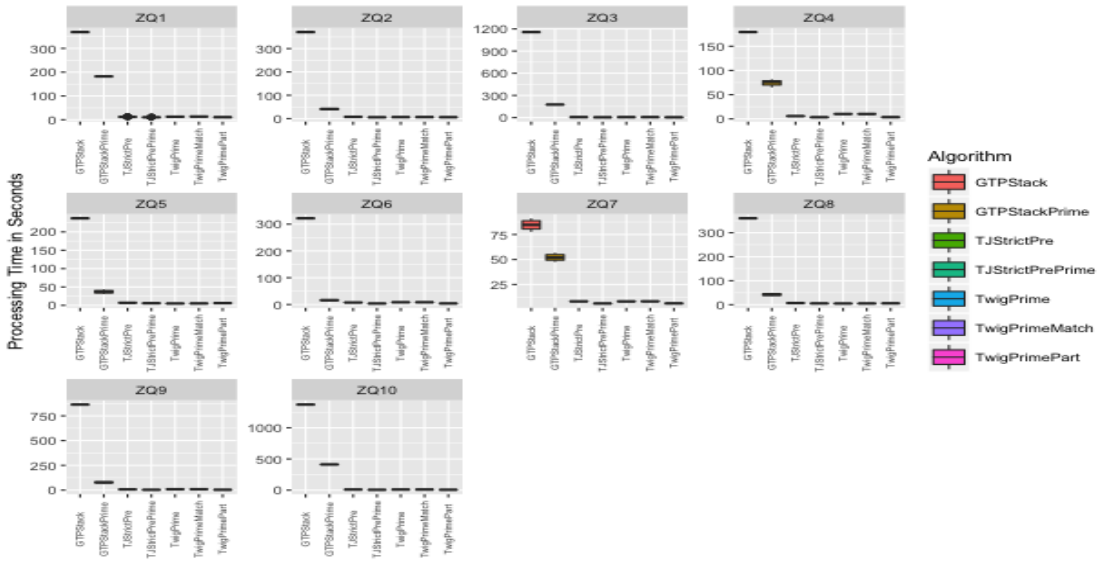


(a) the Zipf queries generated by template 4

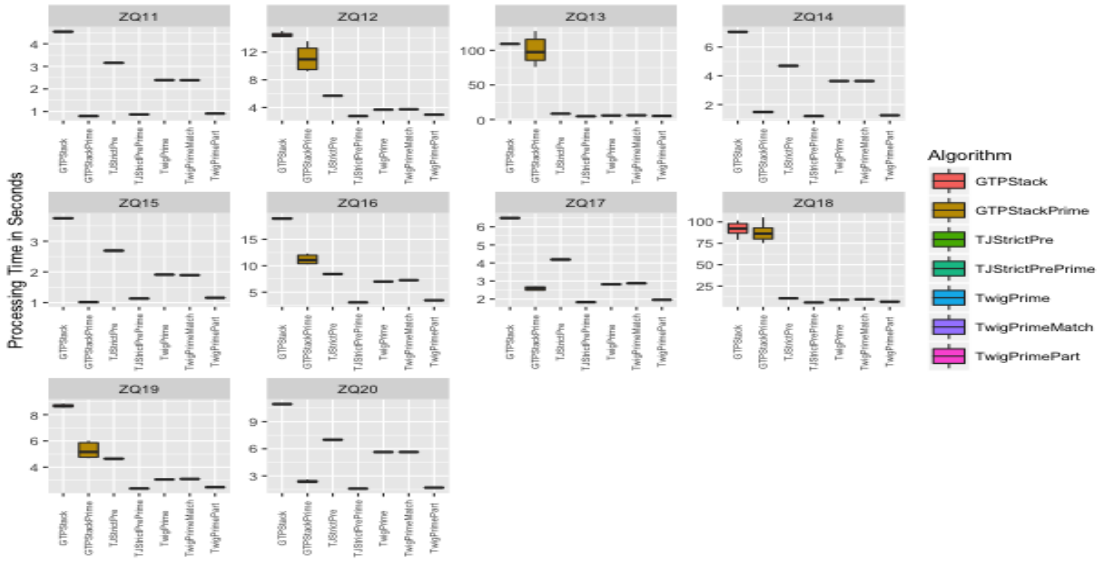


(b) the Zipf queries generated by template 5

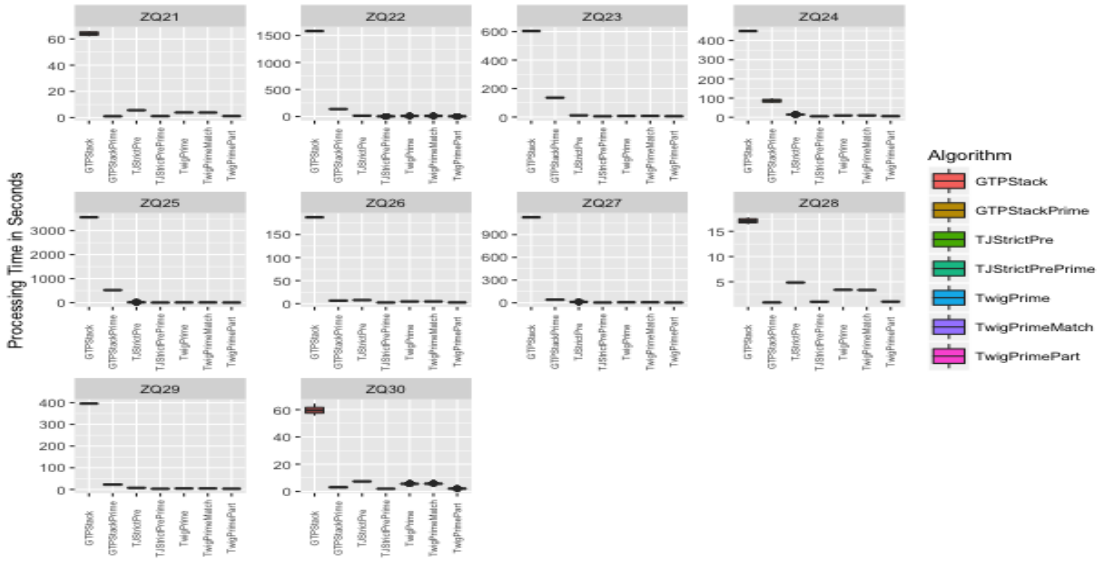
Figure 8.19: The number of elements stored in the intermediate storage by each algorithm for the queries tested over the Zipf dataset. "Actual" represents the number of elements relevant to the query results.



(a) the Zipf queries generated by template 1

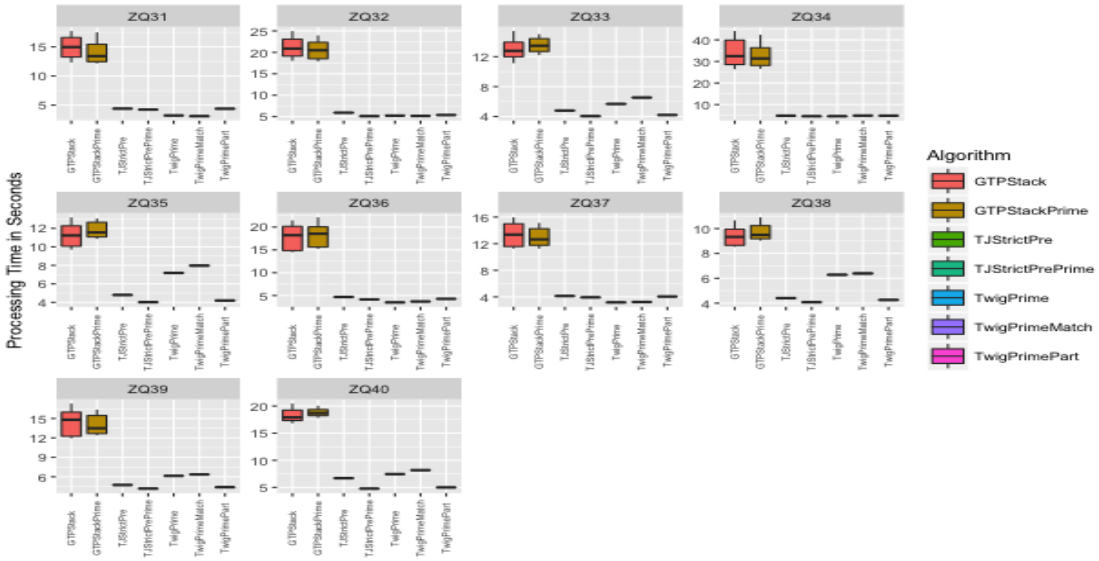


(b) the Zipf queries generated by template 2

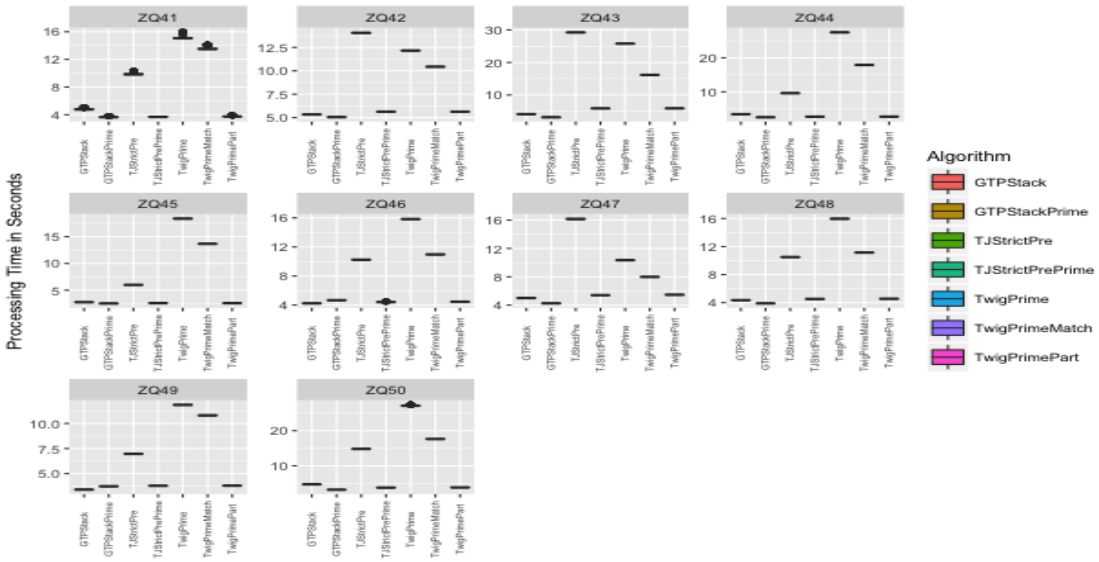


(c) the Zipf queries generated by template 3

Figure 8.20: Query running time of the algorithms against the Zipf dataset.



(a) the Zipf queries generated by template 4



(b) the Zipf queries generated by template 5

Figure 8.21: Query processing time of the algorithms against the Zipf dataset.

8.4.3.6 Scalability

This section aims to simulate and test scalability with respect to the processing time of the new approaches. In this experiment, two datasets were used, the XMark and Random datasets. While the XMark dataset is shallow and data oriented, the random collection has a very recursive structure. Five different versions of XMark were created using the scaling factor from 1 to 5 as explained in Section 5.4.4. The Random dataset was partitioned into 10 different datasets to evaluate the scalability of the algorithms over deeply irregular datasets (see Section 5.4.4). In order to make the experiment comprehensive, two TPQs have been selected over each group of datasets. The "CPL" and "CPL_" approaches were outperformed by the existing algorithms in one of them . Henceforth, XQ_1 and XQ_6 were selected for the XMark datasets as TJStrictPostPrime_ significantly outperformed the other approaches in XQ_1 while TwigFast and TJStrictPost had the best performance for XQ_6. In addition, RQ_5 and RQ_9 were chosen to be issued over the Random datasets because GTPStack performed better than the rest of the algorithms for RQ_5 whereas GTPStackPrime was faster than the other approaches for RQ_9. In order to get meaningful results, algorithms are compared based on the approach used to store the intermediate storage.

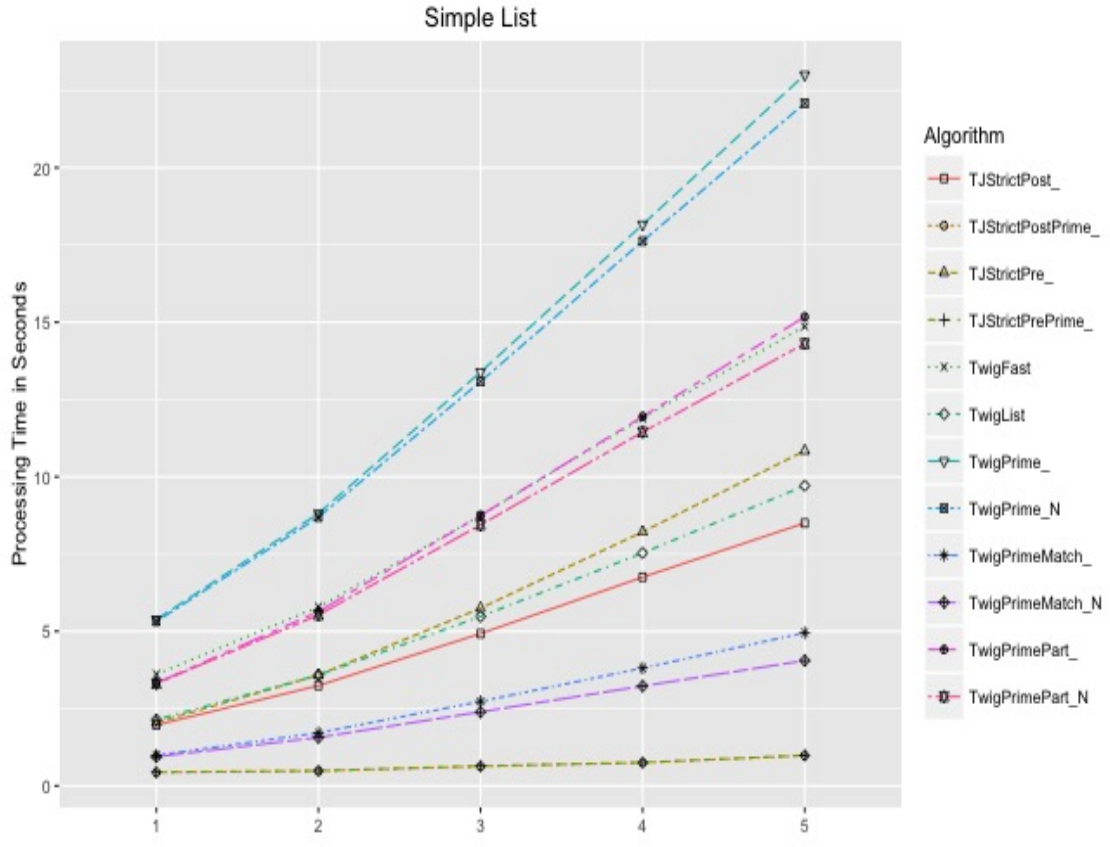
The results for XQ_1 and XQ_6 are illustrated in Figures 8.22 and 8.23. Since the XMark dataset is a data-oriented document with only two recursive tags, it can be observed that algorithms using simple lists scaled linearly with the increasing size of the dataset for XQ_1. On the other hand, when varying the size of the XMark document, algorithms using the level split approach started to increase dramatically with the large datasets compared to algorithms using the combined approach (i.e., the CPL and level split approach), such as TJStrictPostPrime and TJStrictPrePrime. The reason for this may be due to increasing the size of the dataset increases the cost of maintaining and accessing level split lists, hence the processing time is slower as shown in Figure 8.22b.

When processing XQ_6 against XMark datasets, all the algorithms which use the simple list approach showed the same performance, they scaled almost linearly with the increasing size of the dataset (see Figure 8.23a). Closer inspection of the graph shows that TwigPrimeMatch_N significantly outperformed the other approaches. However, the overhead for using linked lists to build the intermediate storage in GTPStack can be observed in Figure 8.23b, although elements stored in a linked list correspond to only one query node. Thus, GTPStack and GTPStackPrime started to increase dramatically with the large datasets compared to the other approaches. Note that all algorithms in the experiments except TwigList can provide optimal evaluation for XQ_6.

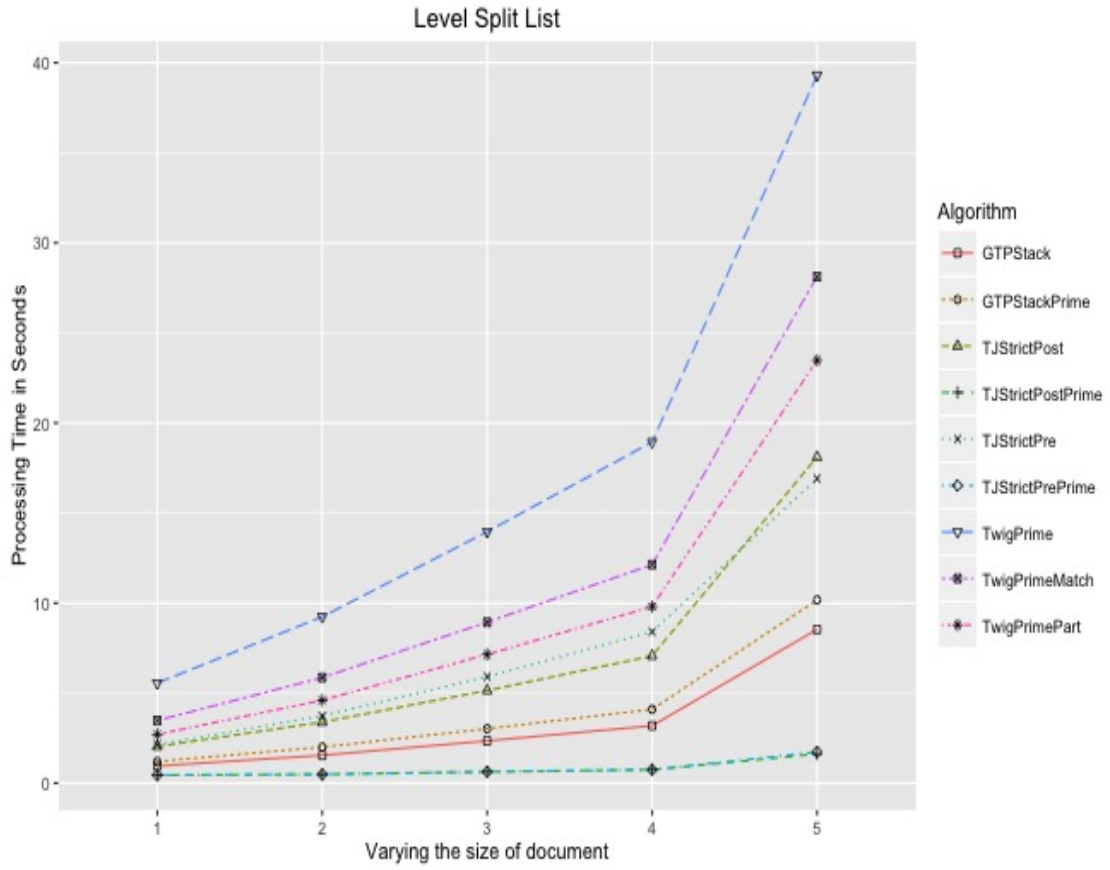
Turning now to evaluate the scalability of the algorithms compared against the Random datasets, the results of scalability analysis for RQ_5 are shown in Figure 8.24. Apart from TwigList, the algorithms scaled effectively and presented a linear relationship with increasing size of the dataset. From this data, algorithms using *getPart* and *getMatch* functions scaled better than algorithms using the *getNext* function (see Figure 8.24a). For

algorithms using the level split approach, it can be seen that all algorithms scaled linearly with the increasing size of the dataset. For RQ_9, it can be seen from Figure 8.25a that the performance of algorithms combining different advanced preorder filtering functions and additional fitering strategies presented a linear relationship with the increasing of the size of the dataset. However, TwigList and TwigFast had the worst scalability for RQ_9 because they only apply a weak subtree filtering check. In addition, algorithms using the level split approach scaled linearly with the increasing size of the dataset with the exception of GTPStack as shown in Figure 8.25b. The observed increase in GTPStack could be attributed to the cost of enumerating results over large intermediate results stored in linked lists. From the data in Figure 8.25b, it is apparent that the benefits of using the CPL approach in GTPStack overcome the overhead of utilising linked lists in the intermediate storage as the best performance was achieved by GTPStackPrime.

To sum up, the experiment considered different structures of TPQs over two groups of different datasets in terms of structural complexity. It can be, therefore, concluded that the new approaches are more scalable than the existing algorithms in processing large datasets.

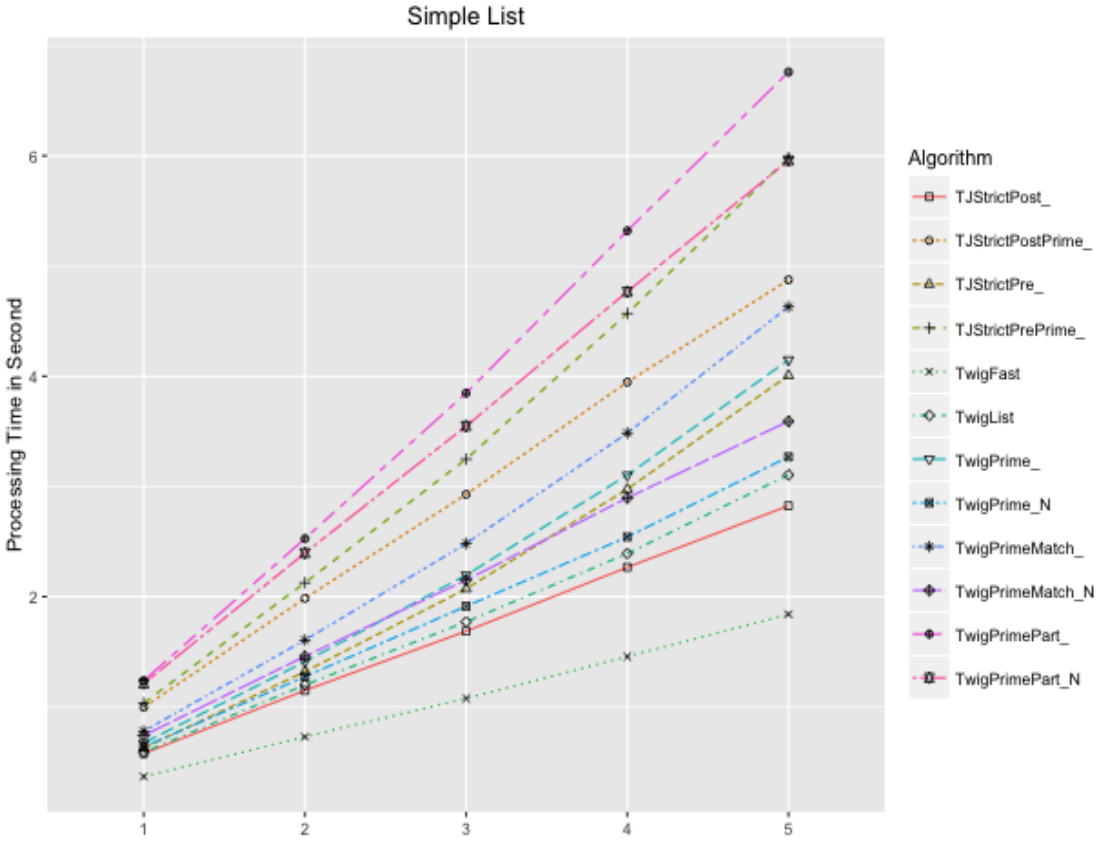


(a)

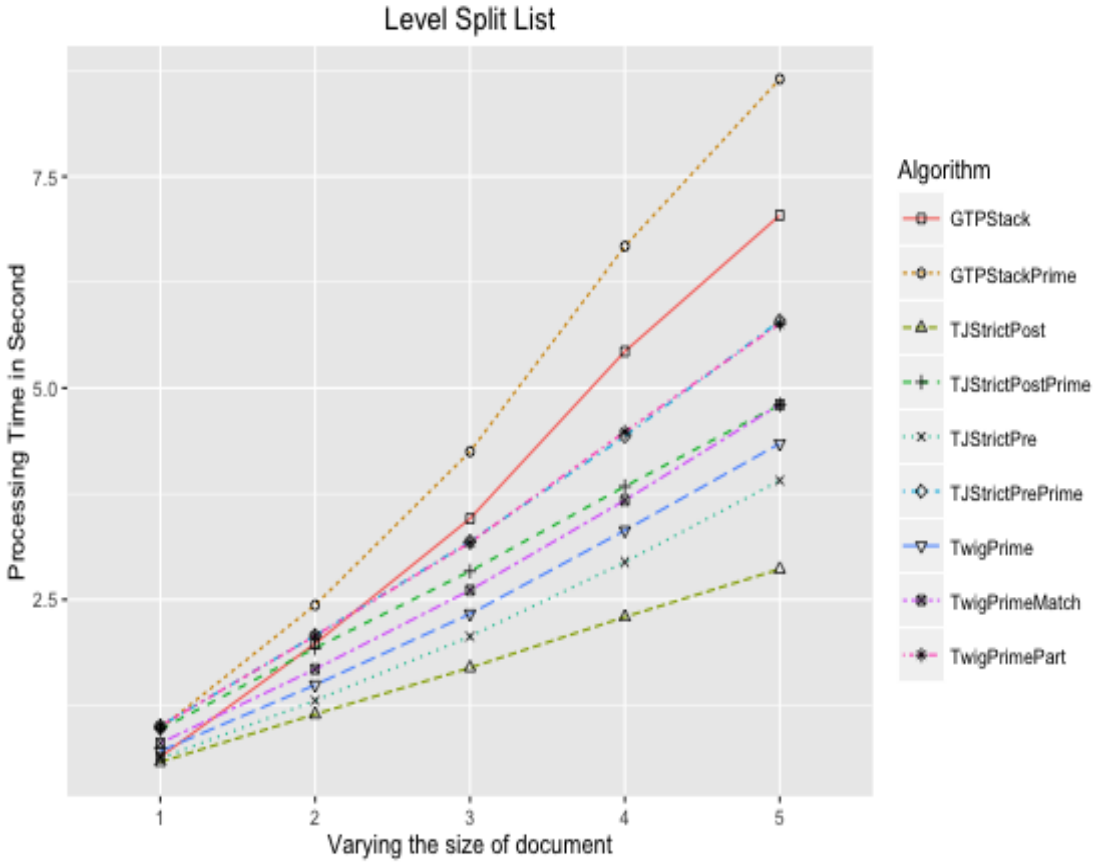


(b)

Figure 8.22: Scalability comparison for XQ_1 against XMark datasets.

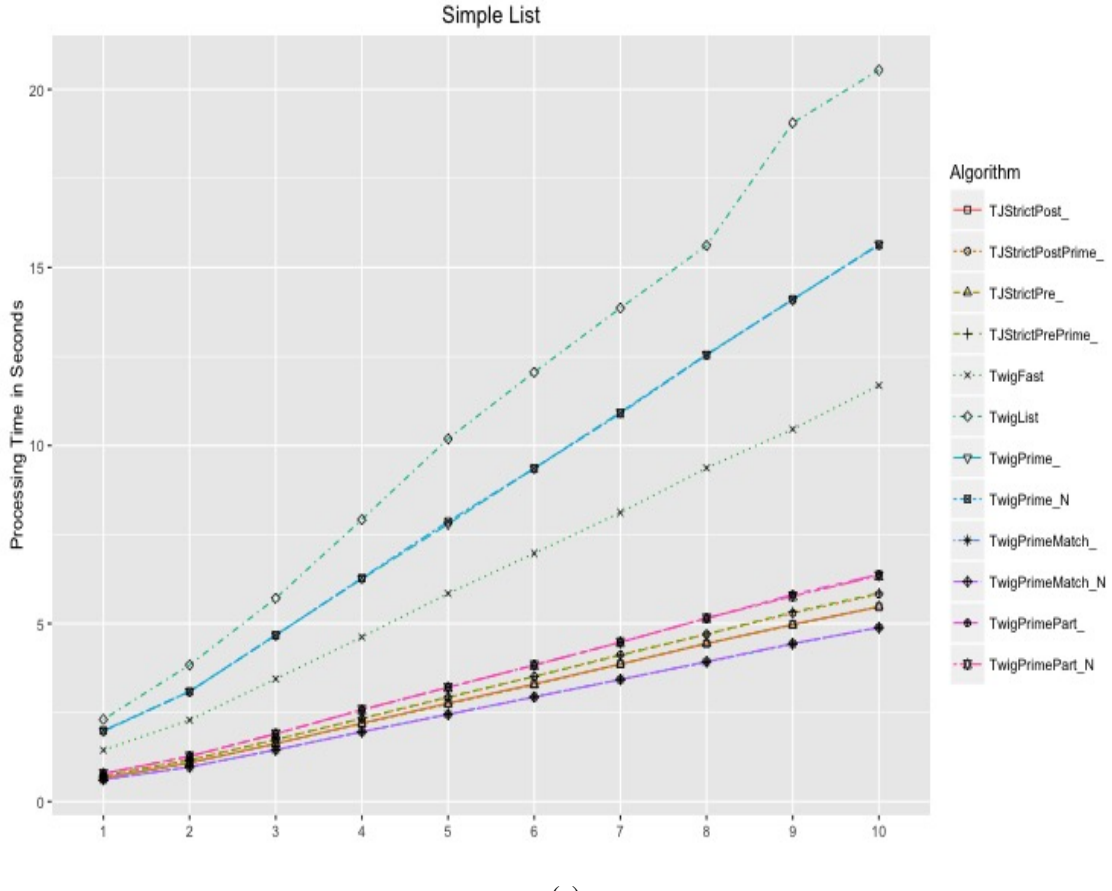


(a)

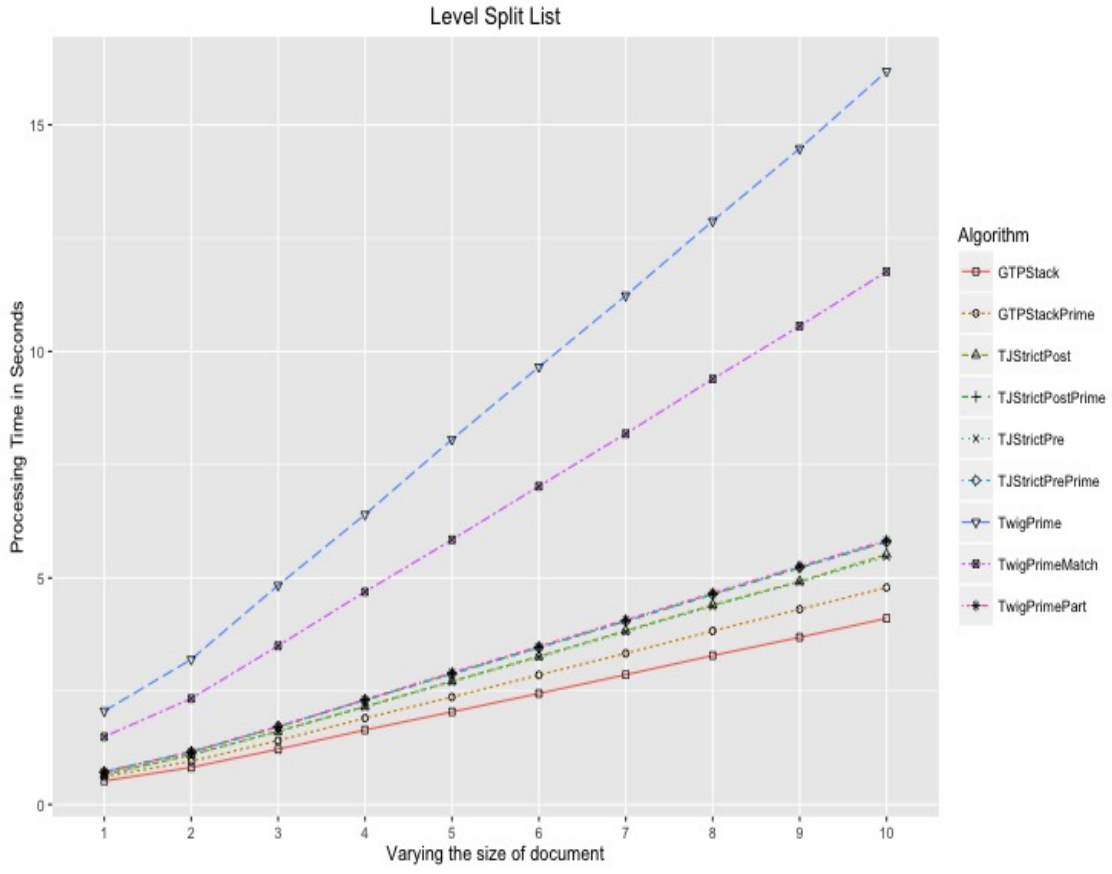


(b)

Figure 8.23: Scalability comparison for XQ_6 against XMark datasets.

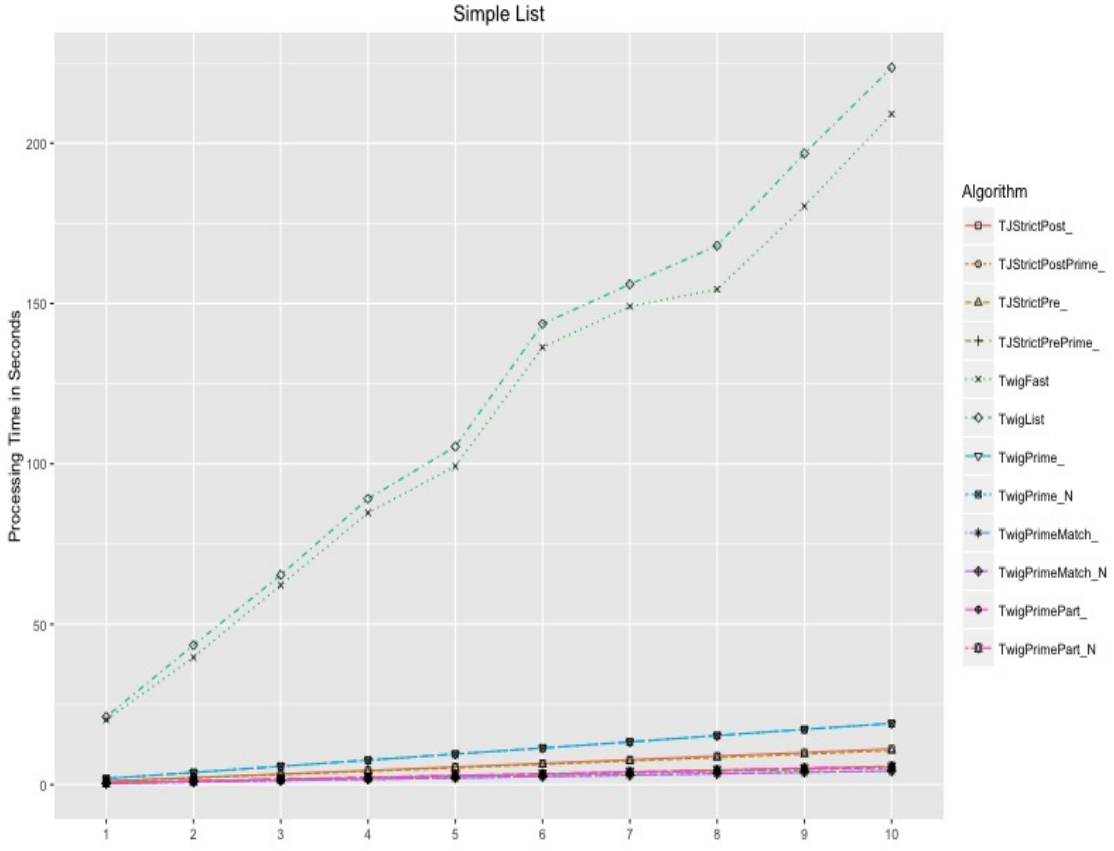


(a)

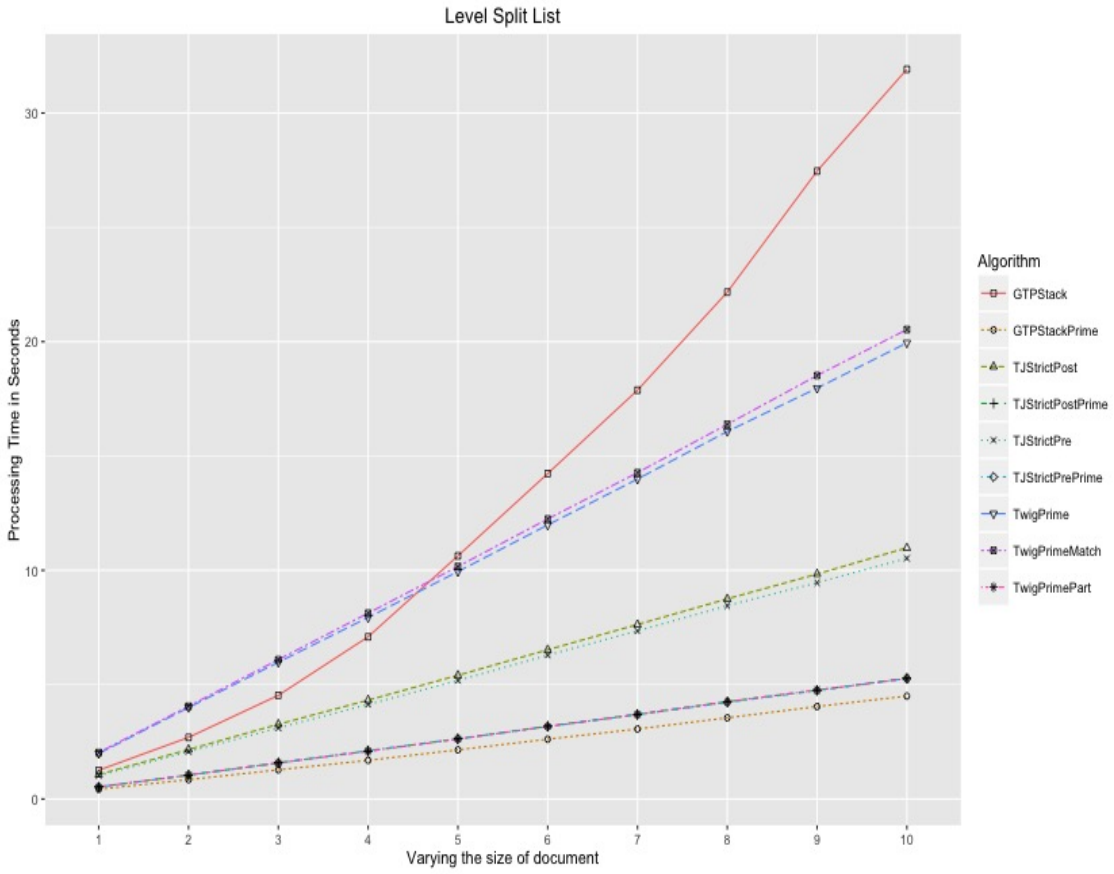


(b)

Figure 8.24: Scalability comparison for RQ_5 against Random datasets.



(a)



(b)

Figure 8.25: Scalability comparison for RQ_9 against Random datasets.

8.4.4 Summary

The aim of the present research was to examine the effects and possible overheads of using the CPL approach in bottom-up holistic twig matching algorithms. The experimental results have shown that the CPL approach can filter out many irrelevant elements effectively and it can be observed that the number of elements stored by the "CPL" and "CPL_" algorithms is always fewer than that stored by the other up-to-date approaches. Furthermore, the CPL approaches have the best performance in comparison to the state-of-the-art algorithms in most cases. The reason for this is that the CPL filtering minimises the cost of building intermediate results because their size is reduced and the cost of enumerating results because unnecessary traversal is avoided. The data reported, in the previous sections, appear to support the assumption that using of the same advanced preorder filtering function and the same design of algorithm (e.g., pointers, local stacks with references) for all TPQs is not always the best approach. In addition, the scalability tests demonstrated that the new approaches can scale well for large datasets. Note that algorithms storing elements in the intermediate storage in postorder may output matching tuples unordered. Eventually, these experiments confirmed that the CPL approach's filtering does not bring any overhead in most cases.

8.5 Conclusion

In this chapter, the research study presented new approaches that use the CPL approach to improve the filtering phase of bottom-up twig matching algorithms. This thesis also introduced a novel design of algorithm which can use the level split approach along with the CPL indexing in TwigFast without maintaining stacks in order to process TPQs efficiently. Furthermore, multiple statistical analysis revealed that the new holistic twig matching algorithms significantly outperformed the existing approaches in terms of the size of the intermediate storage and query running time.

In the next chapter, the advanced preorder filtering strategies will be applied to bottom-up twig matching algorithms to process efficiently ordered TPQs and TPQs with positional predicates by filtering out irrelevant elements with ordered constraints and positional predicates. The advanced preorder filtering functions proposed in this thesis (see Chapter 7) to process ordered TPQs will be added to bottom-up twig matching algorithms in the same way the original *getNext* with the property of CPL was embedded in algorithms proposed in this chapter.

Chapter 9

Ordered and Positional Twig Pattern Matching: Bottom-Up Approach

9.1 Introduction

This chapter describes the task of designing efficient evaluation algorithms for ordered TPQs (OTPQs, for short) and TPQs with positional predicates in a bottom-up twig matching approach. Up to now, no one has proposed an advanced preorder filtering function which can process OTPQs and works properly with bottom-up algorithms with the exception of those proposed in this thesis and introduced in Chapter 7. The efficient selection of useful elements for OTPQs applied by OTJPrimeList and OTJPrime algorithms will be combined with bottom-up twig matching approaches proposed in Chapter 8. To process positional predicates, a novel approach will be proposed to extend the existing preorder filtering functions to skip irrelevant elements which do not satisfy positional predicates using cursor forward movement.

The purpose of this chapter is to discuss how ordered twig pattern queries (OTPQs) can be evaluated in holistic bottom-up approaches. Two approaches are proposed to process OTPQs efficiently which inherit filtering checks for ordered constraints and sequence operators from the advanced preorder filtering functions used by OTJPrimeList and OTJPrime, respectively. In addition, a novel approach which combines preorder and postorder filtering strategies to identify useful elements in TPQs when positional predicates are involved is proposed. Then, the development of a novel holistic twig matching algorithm based on the new filtering approach for processing TPQs with positional predicates efficiently is discussed. Eventually, an extensive set of experiments is conducted to evaluate the performance, scalability and efficiency of the new bottom-up holistic twig matching algorithms to process OTPQs and positional predicates.

The rest of this chapter is structured as follows. Section 9.2 describes some preliminaries including the notation and data structures in one phase holistic algorithms and the limitations of the existing bottom-up algorithms to evaluate sibling axes and positional predicates. Then, the development of new algorithms, OTwigPrimeList to process ordered

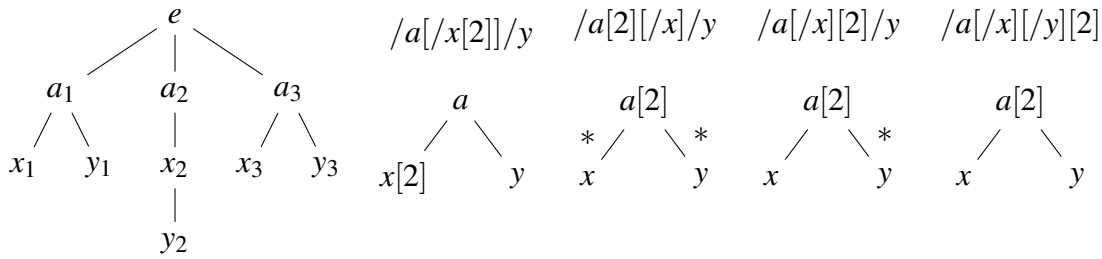
axes, and OPTwigPrime to process OTPQs and positional predicates will be presented in Section 9.3. Section 9.4 describes the experimental evaluation and reports the performance comparison between the algorithms. The chapter will be concluded in Section 9.5.

9.2 Preliminaries

9.2.1 Notation and Data Structure

In the XPath specification [222], positional predicates can be invoked through a pre-defined function called *position()* inside predicates in the form *position() op n*, where *op* is one of the basic logical comparison operators $\{=, <, >, \leq, \geq, \neq\}$ and *n* is an integer. For example, a path expression to find the second x-node which must be a descendant of an a-node which must have a y-node as one of its children can be written as $a[/y]//x[position() = 2]$ or abbreviated as $a[/y]//x[2]$. These have been considered in some limited approaches as in [217, 70]. Evaluation of TPQs with positional predicates takes into account the current context node (i.e., element in the XML tree) and current context-siblings (i.e., elements with the same tag sharing the same parent) as introduced in XPath specification [222]. For illustration, consider the XML tree T_1 in Figure 9.1 and the query Q_2, when the current context element a_3, the current context siblings is a sequence which contains $\{a_1, a_2, a_3\}$, while for the query Q_4 the current context siblings is a sequence which contains $\{a_1, a_3\}$ because a_2 does not have child y-node. The representation of XPath expressions with positional predicates in this thesis is based on the ideas presented in [70]. Structural constraints in TPQs with positional predicates are divided into two types: pre-structural constraints which refer to structural constraints specified prior to the positional predicate and post-structural constraints which identify structural constraints specified after the positional predicate. The different representations modelled as twigs are achieved by labelling post-structural constraints with "*". For example, consider the following query with positional predicate as $Q_1 = //a[//x][2]/y$, the A-D relationship between query nodes *a* and *x* is called pre-structural relationship because it has to be satisfied in order to check the positional predicate, while the P-C relationship between query nodes *a* and *y* refers to as post-structural relationship because it should be checked when the context node *a* has already satisfied both the pre-structural relationship and positional predicate. This classification is due to the fact that positional predicates are not permutable. For instance, $Q_1 = //a[//x][2]/y$ is different from $Q_2 = //a[//x][/y][2]$. That is, the former looks for the second element *a* which has already been found to have x-descendants and y-child, while the latter searches for the second element *a* among its context-siblings which has already been found to have x-descendants and y-child.

Most of the notation and data structures used in this chapter are the same as those in Sections 8.2.1 and 7.2.1. The only exception is that there are extra, auxiliary functions on nodes of TPQs to facilitate the evaluation of TPQs with positional predicates. Supported functions are as follows: *preChildren(q)* returns all child nodes of *q* which



(a) an XML tree T_1. (b) a twig query Q_1. (c) a twig query Q_2. (d) a twig query Q_3. (e) a twig query Q_4.

Figure 9.1: A sample of an XML data tree and twig pattern queries with positional predicates. The edges associated with the positional predicates are unlabelled while edges labelled with "*" should be checked after satisfying the positional predicate.

have structural relationships with q before the positional predicate. *preChildrenAD(q)* returns all child nodes which have A-D relationship with q before the positional predicate. *preChildrenPC(q)* returns all child nodes which have P-C relationship with q before the positional predicate. *hasPP(q)* returns a boolean value to identify whether q has a positional predicate or not. *getPPValue(q)* returns the numerical value of the positional predicate of a query node q. *getPPOperator(q)* returns the relational operator of the positional predicate of a query node q. In addition to a stack for each query node in order to perform a strict prefix path matching used in TJStrictPrePrime as an extension to TJStrictPre, the approach proposed in this thesis uses a *hashtable* to keep track of the position of the context element. This is achieved by using the *parentID* attribute that is maintained in the positional representation of each element as a 5-tuple *(start,end,level,parentID,CPL)*. The set of keys in the *hashtable* is a set of *parentID* values for context elements corresponding to query nodes with positional predicates. The common *hashtable* operations, such as *get(key), put(key, value) and replace(key, value)*, are used.

According to the XPath specification [222], there are four typical cases that the positional predicate may appear in a TPQ. The following four queries exemplify these four representative cases: $Q_1 = //a[/x[2]]/y$. $Q_2 = //a[2][/x]/y$. $Q_3 = //a[/x][2]/y$. $Q_4 = //a[/x][/y][2]$. The twig representations of these queries are depicted in Figure 9.1. In the first case, Q_1, the positional predicate is associated with a leaf query node. That is, only the second element x must be considered as the mapped element for each element a. The rest of queries show cases where positional predicates are used to restrict parent query nodes. For Q_2, the query looks for the second a element if it has child x-element and y-element. This is similar to the first case in that only the second element a must be considered as the mapped element in the XML tree. The third case is considered the most challenging query because it asks for the second element a among its sibling elements which has already satisfied the pre-structural constraints. After that, the mapped elements must also satisfy the post-structural constraints. In the last case, the query searches for the second element a among its siblings which has satisfied the pre-structural constraints.

The next subsection will discuss the limitations of the existing bottom-up matching algorithms to process *following-sibling* and positional predicates. Simple examples will be presented to demonstrate the potential benefits of applying a combination of preorder and postorder filtering to improve the performance when processing TPQs which may contain ordering and positional constraints.

9.2.2 Motivation

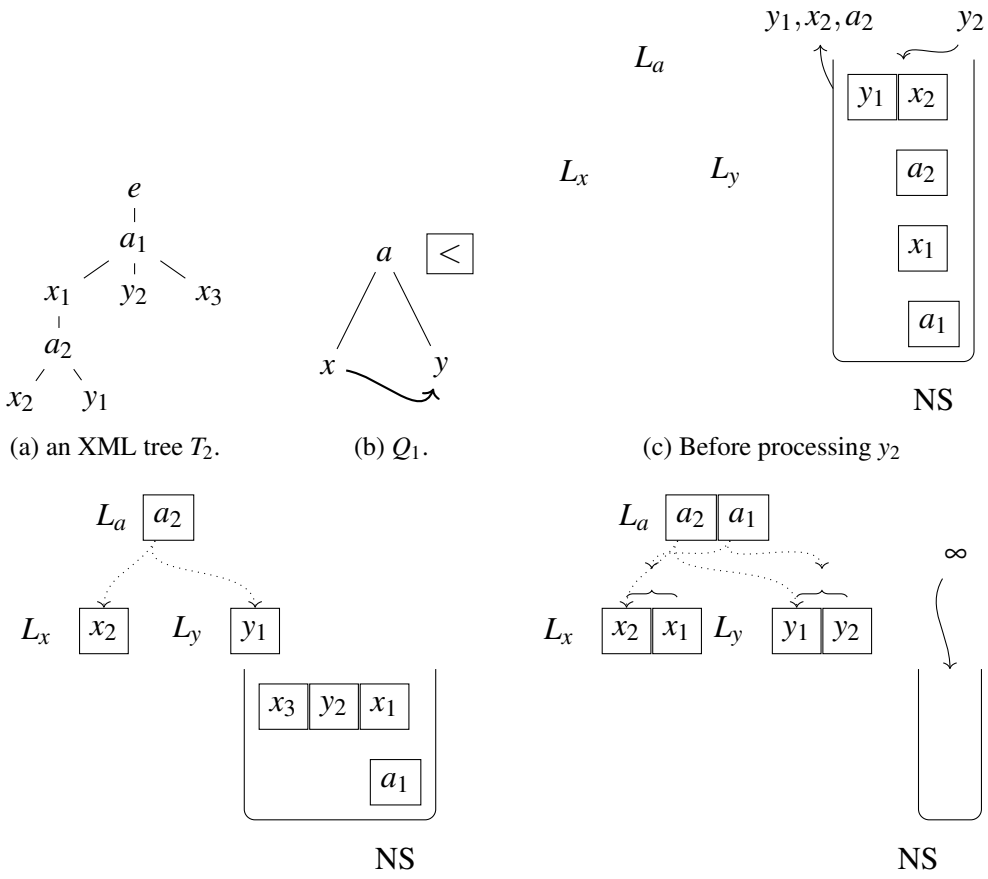
XML query languages such as XPath and XQuery support functions in addition to the basic thirteen axes introduced in the specification of XPath. The positional predicate or function is commonly used to increase meta-data of TPQs. Previous studies of XML twig join processing have not dealt with positional predicate in advanced preorder filtering functions. Most studies of positional twig join processing have only been carried out in the original twig join algorithm such as Stack-tree and TwigList. The authors in [217] extended the binary structural joins algorithm, Stack-tree to support XML TPQs with positional predicates and process *following-sibling* axes. In their approach, every element (i.e., an ancestor candidate) in the stack is associated with *counter* to check whether a new element with a positional predicate can be joined or not. For processing *following-sibling* edges, a list called *Context Sibling List, for short CSL* is associated with every entry (i.e., the *parentID value*) in the stack to check whether a new element has useful a *preceding-sibling* element or not. However, the key problem with this approach is that it generates large intermediate results even when the set of query matches is small. This is due to decomposing the query into binary relationships. One important point to note here is that this approach does not fall within the class of holistic twig matching algorithms.

In [70], the first holistic twig matching algorithm, called TwigPos which can process positional predicates and *following-sibling* relationships was proposed. It is an extension to TwigList algorithm which replaces global stack with nested stacks in order to maintain *following-sibling* constraints among document elements and provide information about current context-siblings. This novel mechanism does not improve the performance of holistic twig matching algorithms rather than providing a strict strategy to check *following-sibling* relationships and positional predicates. Although, the TwigPos algorithm can avoid storing useless elements with *following-sibling* relationships in the intermediate storage, the number of elements stored are huge because it does not use any advanced preorder filtering function. Thus, its processing time is not influenced by the query result [22]. TwigPos can support TPQs with positional predicates where pre-structural constraints are a mix of P-C and A-D relationships, while post-structural constraints are a combination of P-C, A-D and *following-sibling* relationships. Figures 9.2 and 9.3 illustrate the matching process of TwigPos. The key idea is to check pre-structural constraints and positional predicates before a new sibling element is pushed into the corresponding inner stack since the current element is accessed before any of its following siblings and context siblings in document order. As a result, post-structural relationships which may contain *following-*

sibling edges can be checked when the current element is popped up from the nested stack. That is, every *following sibling* element has to scan over the corresponding stack to mark elements which satisfy the *following-sibling* relationship. In the same way, when a new element associated with positional predicate is about to be pushed into the corresponding stack it has to traverse over the corresponding stack to set its position among context sibling elements. This would take $O(w)$ in the worst-case, where w is the maximum degree of element in the original XML tree. The main weakness with TwigPos is that elements are appended to the intermediate results in postorder. Thus, the enumeration process outputs query matches unordered. Unlike TwigList in which some elements may be appended to the intermediate lists even though they do not satisfy P-C relationships, TwigPos performs a strict subtree filtering check when TPQs contain P-C edges by traversing intervals of the current element. This is inefficient because it may take $O(d \times w)$ in the worst-case where d is the longest path in the original XML data and w is the maximum degree of element in the original XML tree. The reason for this is that pre-structural constraints must be satisfied before proceeding to check positional predicates in order to correctly return answers to TPQs. In other words, the algorithm must be able to maintain context sibling elements with the existence of pre-structural constraints. To achieve this, the authors proposed a mapping table to track the number of elements which do not satisfy pre-structural constraints using the parent query node stored in the nested stack as key. Therefore, an element satisfies the positional predicate if its position among its sibling elements with the same tag satisfies the numerical relational operator specified by the positional predicate to the integer value of the positional predicate plus the number of sibling elements which do not satisfy pre-structural constraints. For example, when processing Q_3 in Figure 9.1, a_3 is found to be a match for the query because a_2 failed to satisfy the P-C pre-structural constraint with y-node so that a_3 satisfies the positional predicate since $3 = 2 + 1$, where 2 is the integer value of the positional predicate and 1 indicates the number of elements which are useless (i.e., a_2).

The key problem with the previous approaches [217, 70] is that they can only support TPQs with positional predicates which are on P-C edges. As noted in the original description [70], unnecessary scans over in-memory data structures may be performed to set elements' positions, mark useful elements and filter out irrelevant parents, but further changes are needed to improve the overall evaluation. Another problem with the existing approaches is that they fail to take the semantics of ordered and sequence relationships in twig-based algorithms into account (see Chapter 7).

In view of the success and limitations of the previous approaches in [217, 70], this research study proposes a new holistic bottom-up matching approach to process TPQs with positional predicates which focuses on reducing the overhead of storing irrelevant elements and performing redundant computations. The new approach can support TPQs with positional predicates which are associated with query nodes under both P-C and A-D edges.

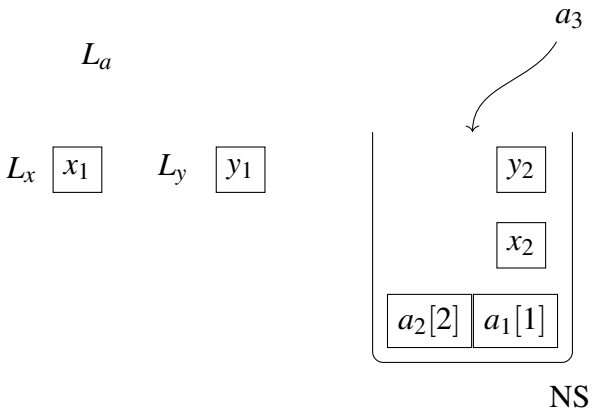


(a) an XML tree T_2. (b) Q_1. (c) Before processing y_2

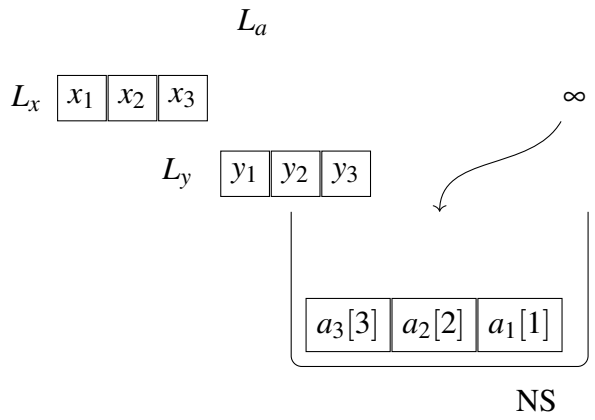
(d) When processing x_3 (e) The intermediate results with intervals

Figure 9.2: Illustration of TwigPos processing *following-sibling*.

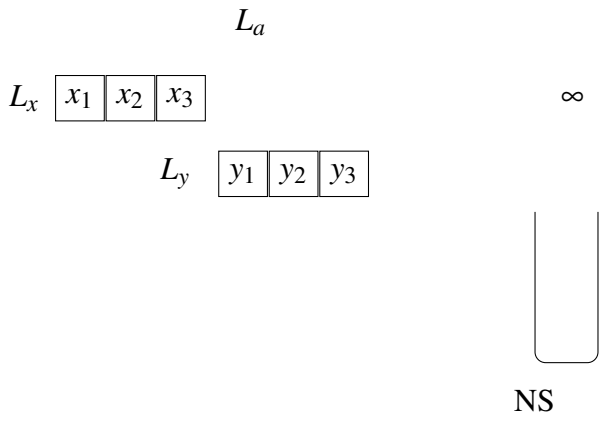
The next section presents a set of new holistic bottom-up twig matching algorithms which combine the efficient selection of useful elements for ordered TPQs introduced in Chapter 7, the level split data structure for strict subtree checks and the extra filtering pass introduced in [89]. In addition, a new technique which provides the accurate and efficient solution to support positional predicates in holistic bottom-up twig matching will be introduced.



(a) Before processing a_3



(b) After processing all elements in streams



(c) The intermediate storage is ready for enumeration

Figure 9.3: Illustration of TwigPos processing positional predicate. Evaluation of Q_3 against T_1 of Figure 9.1.

9.3 Child Prime Label Approaches to support Ordered and Positional TPQs

In this section, the thesis introduces a novel set of bottom-up twig matching algorithms which can combine the advantages of different techniques [132, 89, 22, 12] to match ordered TPQs and TPQs with positional predicates. To process ordered TPQs, OTwigPrimeList can be seen as a combination of the efficient selection of useful elements for ordered TPQs used by OTJPrimeList introduced in Chapter 7, the design of TJStrictPrePrime as the main algorithm and the *getMatch* function as the advanced preorder filtering strategy. The reason for this is that the experimental results in Chapter 8 serves as a hint for the selection of the most appropriate combination to improve query efficiency in bottom-up algorithms. When positional predicates are present, OPTwigPrime is introduced which provides a general and efficient solution to support OTPQs and TPQs with positional predicates. OPTwigPrime extends the *getMatch* with the CPL approach to consider the minimal ordered constraint which does not affect the query answers. It can be seen as alternative to OTwigPrimeList without the buffering technique when the queries containing ordered axes, sequence operators and positional predicates.

The next subsections describe the novel approaches and introduce a new approach to provide efficient filtering for positional predicates into the holistic approaches.

9.3.1 Ordered Bottom-Up Twig Matching Algorithm

The improvement of OTJPrime and OTJPrimeList introduced in Chapter 7 can be trivially ported to the up-to date algorithms. In this thesis, OTwigPrimeList is introduced which is a less involved modification of the original algorithms TJStrictPrePrime and TwigPrimeMatch proposed in this thesis (see Chapter 8). This is due to the fact that it is based on an advanced preorder filtering strategies (i.e., *getPart* and *getMatch(q)*) which are extensions of *getNext*. To achieve the above improvement, *getPart* and *getMatch* using the CPL approach are augmented with the buffering technique introduced in Chapter 7 to process OTPQs efficiently. Consequently, *getPart(q)* and *getMatch(q)* return an element e_q of a query node $q \in OTPQ$ with five properties. The first four properties are inherited from the ordered child and descendant extension introduced in Chapter 7 while the fifth property aims to check the ordered extension introduced in Chapter 7.

i e_q has a descendant element e_{q_i} in each of the streams corresponding to its child elements where e_{q_i} is the head element of a query node $q_i \in children(q)$.

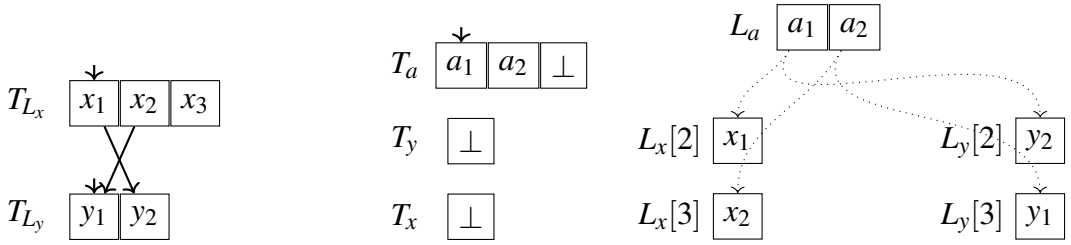
ii each of its child elements satisfies recursively the first property.

iii if q has Parent-Child edge(s) with its child query nodes, then e_q has a child e_{q_i} in T_{q_i} for each query node $q_{q_i} \in childrenPC(q)$ (this property is checked by *getElement* function introduced in Chapter 6).

iv if q has child nodes with ordering constraints, then each of its child elements with the ordering constraints has the ordered extension according to Definition 7.6. This property is newly introduced to *getPart* and *getMatch* proposed in Chapter 8.

v if $\neg isRoot(q)$, then e_q has a relevant ancestor e_p which has been the head element of a query node p = parent(q) in a previous call of *getPart(p)* (*getMatch(p)*, respectively).

Figure 9.4 illustrates the basic ideas of OTwigPrimeList when evaluating OTPQs. It provides an overview of how the OTwigPrimeList algorithm improves the filtering strategy of the existing bottom-up twig matching algorithms by eliminating useless elements which violate the ordering relationships.



(a) Buffering elements with ordering constraints by extraFiltering(a) function.

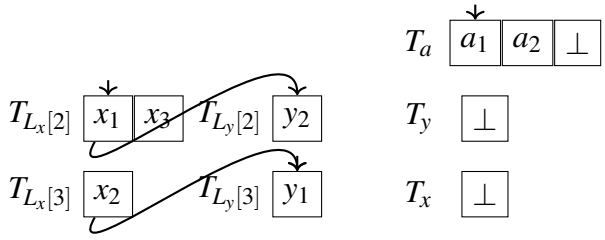
(b) The intermediate results with intervals

Figure 9.4: An example to explain the basic ideas of OTwigPrimeList when processing Q_1 against T_2 in Figure 9.2.

In comparison with TwigPos which outputs matching tuples unordered and does not take into account the constraints capturing by ordered aware TPQs, the effect of OTwigPrimeList can be illustrated in the following example.

Example 9.1. *Consider the XML tree T_2 in Figure 9.2 and the OTPQ Q_1 a/x/following-sibling::y. At the beginning of query processing the head elements are $C_{q_a} \rightarrow a_1$, $C_{q_x} \rightarrow x_1$ and $C_{q_y} \rightarrow y_1$. At the first place, OTwigPrimeList has to assure that the parent query node must satisfy the child and descendant extension and if it has a child query node with ordering constraints, it has to have the ordered extension as well. Hence, the element a_1 satisfies the four properties so that all elements in the T_x and T_y streams which are descendants of a_1 are buffered to their lists for further investigation. Next, the algorithm visits all elements in the buffering lists, T_{L_x} and T_{L_y} to eliminate useless elements. Since elements are found to satisfy the ordered relationship as specified by the query, the algorithm appends elements to their corresponding lists in the intermediate storage in preorder as follows: a_1, x_1, a_2, x_2, y_1 and finally y_2. Consequently, each element in the list has child intervals correctly set as presented in Figure 9.4. Finally, OTwigPrimeList returns the query result consisting of two matches (a_1, x_1, y_2) and (a_2, x_2, y_1). In contrast, TwigPos outputs two matches which are not ordered in their common prefix as (a_2, x_2, y_1) and (a_1, x_1, y_2). It can be seen that during the enumeration process of a_1 TwigPos has to perform unnecessary visits for x_2 and y_1. This is due to the fact that TwigPos uses the simple list approach to store the intermediate results.*

In a like manner, OTwigPrimeList can be extended to avoid unnecessary scans when evaluating *following-sibling* relationships. The original OTwigPrimeList remains the same with the only difference being in using the level split approach to store buffering elements. This is illustrated in Figure 9.5. Note that the output enumeration algorithm is adjusted to consider the ordering relationships as specified by the ordered query. To put it another way, the current element corresponding to a query node with ordering constraints or sequence operators during the enumeration process must have a relevant element falling in the range of its parent intervals. The next section introduces a new approach which can filter out several useless elements when positional predicates are present based on a combination of novel techniques.

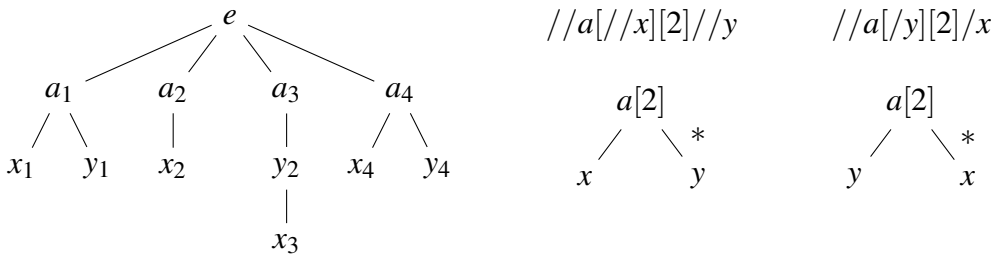


(a) Buffering elements with ordering constraints by extraFiltering(a) function.

Figure 9.5: An example of OTwigPrimeList using the level split approach to buffer elements for Q_1 against T_2 in Figure 9.2.

9.3.2 Ordered and Positional Bottom-Up Twig Matching Algorithm

Matching TPQs with positional predicates to XML data requires the appropriate handling of pre-structural constraints, the positional predicate constraint and post-structural constraints. The existing advanced preorder filtering functions can not naïvely support TPQs with positional predicates since the filtration may result in false negatives because these different types of constraints can conflict: a match that satisfies the pre-structural constraints may violate post-structural constraints, and vice versa. The order in which constraints must be satisfied is very important in order to return query answers correctly, especially when query nodes are associated with both pre-structural constraints and post-structural constraints. For illustration, consider running the original *getMatch* for the query $Q_1 = /a[//x][2]//y$ against the XML tree T_3 in Figure 9.6. The first three calls of getMatch(getRoot(Q_1)) return the sequence a_1, x_1, y_1 because a_1 satisfies the four properties (see Section 9.3.1) so that elements in the subtree rooted from a_1 are returned. The next call will discard a_2 because it does not satisfy the first property (i.e., head elements of child streams must be descendants of a_2). Although a_2 should be considered for further processing since it satisfies the pre-structural constraint, the original *getMatch* returns the incorrect match a_3, x_3, y_2. To overcome this, TwigPos is based on the simple preorder filtering strategy



(a) an XML tree T_3. (b) a twig query Q_1. (c) a twig query Q_2.

Figure 9.6: Hard case with the original *getMatch* and the CPL approach to support positional predicates with a combination of pre-structural and post-structural constraints.

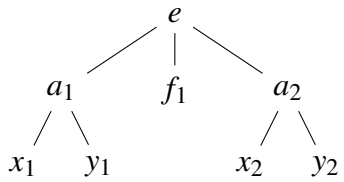
introduced in TwigList [185] which returns elements from the query root down to the query leaf by iterating through the head elements in sorted order of their start values. Thus, a_2 is returned and its positional is found to satisfy the positional predicate since its position equals to the numeral value of the positional predicate plus the number of mismatching sibling elements, 2 = 2 + 0 (a_1 was previously found to satisfy the pre-structural constraint). Similarly, for Q_2 in Figure 9.6, the CPL approach discards a_3 because it does not satisfy the third property (i.e., a_3 must have a child e_{q_i} in T_{q_i} for each query node $q_{q_i} \in$ childrenPC(a)). However, a_3 should be stored for further processing to check the post-structural constraints since it has satisfied the pre-structural and positional constraints as a_3 has y_2 as one of its children and 3 = 2 + 1, respectively. Even though there is no match for Q_2 in the XML tree T_3, the CPL approach returns the incorrect match a_4, y_4, x_4.

The above examples illustrate how the failure to consider the order and type of constraints imposed by positional predicates result in returning irrelevant results by the existing advanced preorder filtering functions. Basically, the incorrect match mainly comes from the first and third property because they do not consider the pre-structural and post-structural constraints when positional predicates are associated with branching query nodes. Since there are four typical cases where positional predicates may appear in a TPQ (see Section 9.2.1), the challenge is to handle the existence of pre-structural constraints in the third and fourth case because the first and second cases are straightforward, and a simple filtering technique over the input streams can be used to discard useless elements. The key idea is to extend the existing advanced preorder filtering strategies to support the pre-structural and post-structural constraints effectively. In addition, the extra filtering pass which performs strict subtree matching will be used to update the number of mismatching siblings and filter out useless elements with respect to the pre-structural, positional and post-structural constraints. The intuition is simple because the use of extra filtering pass with the level split approach can guarantee optimal enumeration so that any combination of P-C and A-D relationships imposed as pre-structural constraints can be checked during a single traverse over the intermediate results of the parent query nodes [89]. It should be noted that when a positional predicate on a query node with a combination of pre-structural and post-structural constraints has preceding, *preceding-sibling* relationships or is pointed to

by sequence operators, using the ordered extension introduced in Chapter 7 gives errors. This is due to the fact that in this case elements must satisfy the pre-structural constraints to update the number of mismatching siblings correctly, and they can not be filtered out before checking whether or not they satisfy the pre-structural constraints. The next example, Example 9.2, shows why using the ordered extension used by OTJPrime (see Chapter 7) gives problems when elements with positional predicates violate the ordered extension. This can also be used to justify not using the buffering technique to support positional predicates.

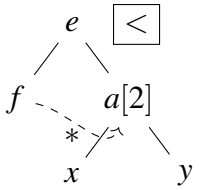
Example 9.2. *Consider the XML trees T_4 and T_5 of Figure 9.7, and the following ordered and positional queries Q_1 = e//a[/x][/y][2]/preceding::f and Q_2 = e//a[/x][/y][2]/following::f. For Q_1 over T_4, if the ordered extension must be satisfied between f_1 and a_1 errors may occur, while in Q_2 against T_5 the ordered extension between a_1 and f_1 can be checked safely. The reason for this is that the ordered extension skips efficiently elements violating SeqLR relationships. Initially, the head elements are $a \rightarrow a_1$, $x \rightarrow x_1$, $y \rightarrow y_1$ and $f \rightarrow f_1$. For Q_1, a_1 violates the ordered extension with respect to f_1 because it has a start value less than the start value of f_1, thus it can be discarded safely according to the ordered extension definition. When a_1 is removed from the stream, there is no way to find out whether or not it satisfies the pre-structural constraints to update the number of mismatching siblings so that incoming elements can be assigned correct positions. As a result, the match (e, f_1, a_2, x_2, y_2) will be lost. For Q_2, the ordered extension can be checked without losing matches because elements with positional predicates are not skipped during this check. f_1 is found to violate the ordered extension with respect to a_1 so that the cursor of the query node f is advanced. Now, all elements satisfy the ordered child and descendant extension, and during the output enumeration they must satisfy the ordering relationships as specified by the ordered axes. This leads to no match because a_2 is found to satisfy the pre-structural and positional constraints but violate the post-structural constraints (i.e., it has no following element f).*

As was discussed in Section 9.3.1, *getMatch* is a core function used as the advanced preorder filtering strategy in twig pattern matching. It is used to identify the next element associated with the query node to be processed and advanced in its corresponding stream. A key to the practical performance of *getMatch* using the CPL approach is that elements returned must satisfy the five properties which can prune efficiently a considerable number of irrelevant elements. As query nodes on the basic axes can be checked using the child and descendant extension, the positional child and descendant extension defined in Definition 9.3, aims to take into consideration query nodes with pre-structural and positional constraints associated with branching elements to check whether or not they likely contribute to the final results. In addition, the ordered and positional extension defined in Definition 9.4 extends the ordered extension introduced in Chapter 7 to avoid filtering out elements before checking the pre-structural constraints, as has been seen in Example 9.2. Before proceeding to explain the new advanced preorder filtering function

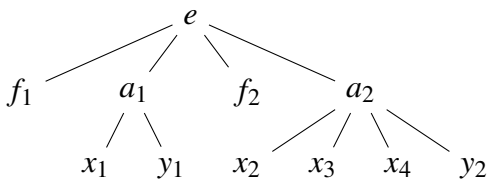


(a) an XML tree T_4.

$e//a[/x][/y][2]/preceding :: f$

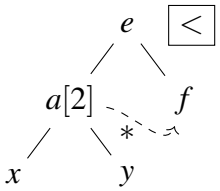


(b) a twig query Q_1.



(c) an XML tree T_5.

$e//a[/x][/y][2]/following :: f$



(d) a twig query Q_2.

Figure 9.7: Problematic case with ordered extension introduced in Chapter 7 and positional predicates

which provides an efficient and general solution to TPQs which may contain ordered and positional predicates, it should be noted that every element stored in the intermediate storage is associated with a positional attribute to test whether or not it satisfies the positional predicate. To achieve this, two hash-based tables are associated with query nodes which have positional predicates. A *hash* table *counter* is used to compute the positional value of each element using the *parentID* value as the key. The second *hash* table *mismatch* is used to record the number of siblings which fail to satisfy the pre-structural constraints and uses the *parentID* value as the key. Accordingly, the procedure *advance()*, which forwards cursors of query nodes by one position to point to next elements, is extended to update the entries of the hash-based tables every time elements with positional predicates are discarded.

Definition 9.3 (Positional Child and Descendant Extension)**.** *A query node q has the positional child and descendant extension if the following properties hold:*

- $\forall\ n_i \in prechildrenAD(q)$, *there is an element* e_i *which is the head of* T_{n_i} *and a descendant of* e_q *which is the head of* T_q.
- $\forall\ n_i \in prechildrenPC(q)$, *there is an element* e_q *which is the head of* T_q *and its CPL parameter is divisible by tagPrime*(n_i).
- $\forall\ n_i \in prechildren(q)$, *if hasPP*$(n_i)$, *then* n_i *must have the positional child and descendant extension, otherwise* n_i *must have the child and descendant extension.*

Definition 9.4 (Ordered and Positional Extension)**.** *A query node q has the ordered and positional extension if the following properties hold:*

- $\forall\, n_i \in rightLR(q)$, *if* $preChildren(n_i) = \emptyset$, *then there is an element* e_i *which is the head of* T_{n_i} *and has a start value greater than the start value of* e_q *which is the head of* T_q.
- $\forall\, n_i \in rightSLR(q)$, *if* $preChildren(n_i) = \emptyset$, *then there is an element* e_i *which is the head of* T_{n_i} *and has a start value greater than the start value of* e_q *which is the head of* T_q.
- $\forall\, n_i \in rightSeqLR(q)$, *if* $preChildren(n_i) = \emptyset$, *then there is an element* e_i *which is the head of* T_{n_i} *and has a start value greater than the start value of* e_q *which is the head of* T_q.

Algorithm 15 shows the general framework for identifying potential elements participating in the final results of TPQs which may have ordered and positional constraints extending *getMatch(q)* proposed in Chapter 8 to take into account Definitions 9.3 and 9.4. Consequently, the branching elements with pre-structural constraints need only to satisfy A-D and P-C relationships specified before the positional predicate to be considered for further processing. In addition, the branching elements must have their children with ordering constraints sorted in ascending order of their start values, except elements with pre-structural constraints. As a result, *getMatch(q)* returns an element e_q of a query node $q \in TPQ$ with four properties. The first and third property are inherited from the child and descendant extension while the second and fourth property aim to check the new definitions introduced in this chapter.

i if q does not have pre-structural constraints, then e_q has the child and descendant extension introduced in Chapter 6.

ii if q has pre-structural constraints, then e_q has the positional child and descendant extension introduced in Definition 9.3.

iii if $\neg isRoot(q)$, then e_q has a relevant ancestor e_p stored in the main algorithm which has been the head element of a query node p = parent(q) in previous calls of *getMatch(p)*.

iv if q has child nodes with ordering constraints, then each of its child elements with the ordering constraints has the ordered and positional extension introduced in Definition 9.4.

The description of the new extension to the original *getMatch()* is as follows: at Line 13, it checks if the current query node has pre-structural constraints. If so, the algorithm gets the *max* and *min* of the head elements corresponding to child query nodes of q which are returned by $preChildren(q)$. Line 16 discards elements of q which do not contribute to the final result considering the positional child and descendant extension. Before proceeding

Algorithm 13: check ordered extension and update counter and mismatch tables

```
1  Function orderedExtensionWithPP(Query node q):
2      foreach node n_i in children(q) do
3          if hasOrderingConstraint(n_i) then
4              foreach node m_j in rightLR(n_i) do
5                  while getStart(getElement(n_i)) > getStart(getElement(m_j)) ∧
                     preChildren(m_j) == ∅) do
6                      if ¬empty(S_q) ∧ S_q.get(0)is not ancestor of C_{m_j} then
7                          advance(m_j)
8                      else
                           return : m_j
9              foreach node m_j in rightSLR(n_i) do
10                 while getStart(getElement(n_i)) > getStart(getElement(m_j)) ∧
                     preChildren(m_j) == ∅) do
11                     if ¬empty(S_q) ∧ S_q.get(0)is not ancestor of C_{m_j} then
12                         advance(m_j)
13                     else
                           return : m_j
14             foreach node m_j in rightSeqLR(n_i) do
15                 while getStart(getElement(n_i)) > getStart(getElement(m_j)) ∧
                     preChildren(m_j) == ∅) do
16                     if ¬empty(S_q) ∧ S_q.get(0)is not ancestor of C_{m_j} then
17                         advance(m_j)
18                     else
                           return : m_j
       return : q
19     // the ordered extension is satisfied
20 Procedure advance(Query node q, Integer c):
21     key = getParentID(getElement(q))
22     if counter_q.contains(key) then
23         temp = counter_q.get(key)
24         temp++ // the corresponding counter is incremented according to sharing
             parents
25         counter_q.replace(key,temp)
26         temp = mismatch_q.get(key)
27         temp++ // the number of mismatching elements increases by one
28         mismatch_q.replace(key,temp)
29     else
30         counter_q.put(key,1)
31         mismatch_q.put(key,1)
32     C_q = C_q + 1
```

Algorithm 14: descendant forward movement and the CPL approach with the pre-structural constraints

```
1  Function descendantForward(Query node q):
2      p = parent(q)
3      if ¬empty(S_p) then
4          if eof(p) then
5              forwardToEnd(q)
6          fwdToDescOf(q,getElement(p))
7      else
8          fwdToDescOf(q,S_p.get(0)) // forward the descendant cursor according to the
             latest ancestor stored in the main algorithm
9          if S_p.get(0) is ancestor of the current element q then
10             if eof(p) then
11                 forwardToEnd(q)
12             fwdToDescOf(q,getElement(p))
13 Procedure fwdToDescOf(Query node q, NodeLabel act):
14     while ¬eof(q) ∧ getStart(getElement(q)) < getStart(act) do
15         advance(q)
16 Function getQCPL(Query node q):
17     // the prime number assigned to the query node which is the product of its child
         query node prime numbers
18     qCPL = 1
19     if preChildrenPC(q) > 0 then
20         foreach node n_i in preChildrenPC(q) do
21             qCPL = qCPL × tagPrime(n_i)
22     else
23         foreach node n_i in childrenPC(q) do
24             qCPL = qCPL × tagPrime(n_i)
       return : qCPL
25 Function getElement(Query node q):
26     if preChildrenPC(q) > 0 ∨ childrenPC(q) > 0 then
27         while ¬ eof(C_q) ∧ getCPL(C_q) % getQCPL(q) ≠ 0 do
28             advance(q)
29     if eof(C_q) then
           return : ∞, ∞, ∞, 1 // out of range label
30         
31     else
           return : C_q // the current head element in the stream of q
32         
```

Algorithm 15: getMatch(q)

Input: q is a query node
Result: a query node in a TPQ with ordered and positional constraints which may or may not be q

```
1  if ¬ isRoot(q) then
2      descendantForward(q) // cursor forward movement according to the latest
         processed element's descendants
3  if isLeaf(q) then
       return : q
4  while true do
5      foreach node n_i in children(q) do
6          if ¬ match[q] then
7              g_i = getMatch(n_i)
8              if g_i ≠ n_i then
9                  match[q] = false
                   return : g_i
10     n_max = a query node with the maximum start value ∈ children(q)
11     n_min = a query node with the minimum start value ∈ children(q)
12     // the following code lines add the positional child and descendant extension for
         child nodes with the pre-structural constraints
13     if prechildren(q) ≠ ∅ then
14         n_max = a query node with the maximum start value ∈ prechildren(q)
15         n_min = a query node with the minimum start value ∈ prechildren(q)
16     while getEnd(getElement(q)) < getStart(getElement(n_max)) do
17         advance(q)
18     // the following code lines return elements with post-structural constraints in
         preorder to perform the simple preorder filtering strategy introduced in
         PathStack
19     if preChildren(q) ≠ ∅ then
20         postNode_min = a query node with the minimum start value ∈ children(q)
21         if postNode_min ≠ n_min then
22             if getStart(getElement(q)) > getStart(getElement(postNode_min)) then
23                 match[q] = false
24                 if ¬empty(S_q) ∧ S_q.get(0) is not ancestor of C_postNode_min then
25                     // pop all elements in S_q and update their end values
26                     C_postNode_min = C_postNode_min + 1
27                     continue // proceed to the next cycle and skipping the following
                         lines
28                 else
                       return : postNode_min
29     if getStart(getElement(q)) > getStart(getElement(n_min)) then
30         match[q] = false
31         if ¬empty(S_q) ∧ S_q.get(0) is not ancestor of C_n_min then
32             // pop all elements in S_q and update their end values
33             C_min = C_min + 1
34         else
               return : n_min
35     else
36         if isOrderedBranching(q) then
37             q_oewpp = orderedExtensionWithPP(q)
38             if q_oewpp ≠ q then
                   return : q_oewpp
39         match[q] = true
           return : q
```

to check whether or not the current element of q satisfies the first and second property, Lines 19-28, ensure that elements which are in post-structural constraints are returned if they precede the head element of q and a have relevant ancestor stored inside the main algorithm, the third property. This novel approach allows the advanced preorder filtering functions to combine the simple preorder filtering strategy introduced in the PathStack algorithm [40] which strictly filters elements according to the node query path from the root and does not perform any cursor forward. It also guarantees that when an element of a query node is returned by *getMatch*, there is no element remaining in the stream which has a start value lower than the start value of the element returned. After that, in Line 29, the first and second property is checked. If the current head element of q fails to satisfy them, the child query node with the smallest start value is checked to see whether or not it has a relevant ancestor stored in the main algorithm. If so, it is returned in Line 34. Otherwise, the current element of n_{min} is discarded and the algorithm proceeds to the next cycle. Next, by getting to Line 36, the current query node q has already satisfied the first three properties of a combination of the positional child and descendant extension. At Line 37, the supporting function *orderedExtensionWithPP* (see Algorithm 13) is called to iterate over children of q to assure that they all have the ordered and positional extension. That is, if a query node, which does not have pre-structural constraints, pointed by LR, SLR or SeqLR edges precedes the source of the edge, it is checked to see whether or not it has a relevant ancestor. If so, it is returned at Line 38. Otherwise, the function advances the corresponding stream to point to the next element until all children of q have the ordered and positional extension. Note that the procedure *advance* introduced in Algorithm 13 keeps track of the number of siblings and mismatching siblings for each inner element. Each time an element is removed from the stream and is determined to be useless, the corresponding *counter* and *mismatch* tables are updated.

Turning now to the main algorithm of OPTwigPrime shown in Algorithm 16, it can be seen as an alternative to OTwigPrimeList, supporting ordered and positional constraints without using the buffering techniques. The key for supporting positional predicates is to choose the appropriate time to skip irrelevant elements which do not satisfy the positional predicate constraint. In OPTwigPrime, there are two time points that are correct for skipping according to where the positional predicate appears in the TPQ (see Section 9.2.1). *Time point 1* is for positional predicates which are associated with leaf query nodes or parent query nodes without pre-structural constraints. Since query nodes belonging to these cases do not have pre-structural relationships, OPTwigPrime immediately skips elements which do not satisfy the positional constraints. As a result, the supporting function *checkPosition()* is called at Line 9 to assure that only elements satisfying the positional predicate are appended to the intermediate storage. *Time point 2* is performed during the extra filtering pass for the third and fourth case in which pre-structural constraints are present. The reason for this is that optimal enumeration can be guaranteed if and only if a combination of the level split and extra filtering pass (i.e., a

strict subtree filtering check) approach is applied for twig matching algorithms building up directly the intermediate storage (see Chapter 8). Accordingly, both contributing and non-contributing elements can be determined during scanning the intermediate results in preorder. At Line 19, the supporting procedure *extraFilteringPass* is used to perform a strict subtree matching and positional filtering check. The branching element must satisfy both P-C and A-D relationships and the positional predicate if it is associated with pre-structural constraints. The previous siblings which do not satisfy pre-structural constraints for each potential element may be visited twice during the filtering phase. This is because OPTwigPrime uses a combination of preorder and postorder filtering. Therefore, the number of mismatching siblings is computed as the number of mismatching siblings in preorder filtering (i.e., *getMatch*) plus the number of mismatching siblings in postorder filtering (i.e., *extraFilteringPass*). In order to achieve this, entries of *mismatch* tables are cleared prior to performing a strict subtree matching at Line 18 of OPTwigPrime. The intuition is simple since elements are processed in preorder in both strategies (see Example 9.5). Once the intermediate storage contains elements with their corresponding child and descendant intervals, Line 20 calls the output enumeration algorithm to enumerate the output. It should be noted that the output enumeration algorithm is extended to consider the ordering relationships as specified by the query. That is, the current element corresponding to a query node with ordering constraints or sequence operators during the enumeration process must have a relevant element within the range of its parent intervals.

Example 9.5. *Consider the XML tree T_6 in Figure 9.8 and the query $Q_1 = //a[/x][/y/f][1]$. When the current element of query node a is a_3, the number of mismatching siblings is 0 because during the process of getMatch both a_1 and a_2 have the positional child and descendant extension with respect to subtrees rooted from a_1 and a_2, respectively. Moreover, y_1 is found to have the child and descendant extension with respect to f_1. However, during the extra filtering pass a_1 is deleted because it does not have element y. Thus, the number of mismatching siblings for a_1 is increased. a_3 is found to satisfy the pre-structural and positional constraints as $a_3.position = 2 = 1 + 1$. Finally, the query result consists of two matches (a_2, x_2, y_1, f_1) and (a_3, x_3, y_3, f_2).*

Unlike previous approaches [217, 70], the purpose of *counter* table is to support the process of positional predicates on A-D edges using the *parentID* value. This is due to the fact that holistic twig matching algorithms are based on partitioning index so that labelling schemes without *parentID* values do not suffice to determine whether or not a specific pair of *a* and *d* elements satisfy a//d[2]. To illustrate this, consider the XML tree T_7 in Figure 9.9 and the previous query $Q_1 = a//d[2]$. In the partitioning approach only elements corresponding to query nodes are scanned so that there is no way to determine sibling relationships between context elements using the original range-based labelling schemes. Without the *parentID* value, two matches of Q_1 against T_6 will be missed. Accordingly, the *parentID* value must be maintained while processing the input sets of *a* and *d* elements to return answers to TPQs with positional predicates correctly. Before proceeding to show

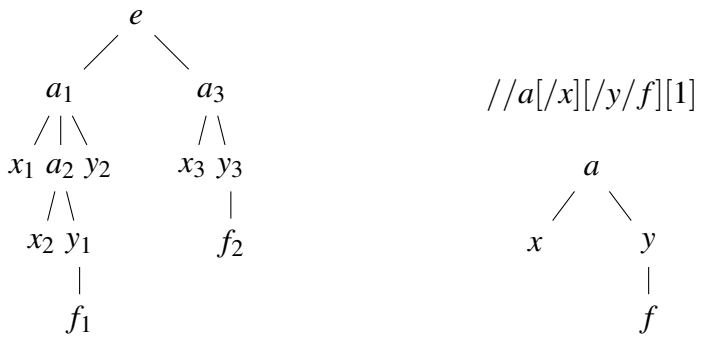
Algorithm 16: OPTwigPrime

```
   Input: TPQ Q
1  foreach q_i ∈ Q do
2  |  match[q_i] = false
3  q_act = getMatch(getRoot(Q)) // see Algorithm 15
4  v_act = getElement(q_act)
5  while ¬end(getRoot(Q)) do
6  |  if ¬ isRoot(q_act) then
7  |  |  processLocalDisjoint(v_act, parent(q_act))
8  |  processLocalDisjoint(v_act,q_act)
9  |  setPositional(v_act,q_act)
10 |  if v_act has a prefix path matching then
11 |  |  if ¬hasPP(q_act) ∨ (preChildren(q_act) ≠ ∅ ∨ checkPosition(v_act, q_act)) then
12 |  |  |  // ∀ n_i ∈ children(q_act) v_act.start_{n_i} = length(getVector(n_i))
13 |  |  |  push v_act into S_{q_act}
14 |  |  |  append v_act to the corresponding intermediate list
15 |  |  C_{q_act} = C_{q_act} + 1 // the number of siblings has been already increased by
   |  |    setPositional()
16 |  |  q_act = getMatch(getRoot(Q)) // see Algorithm 15
17 |  |  v_act = getElement(q_act)
18 ∀q_i ∈ Q ∧ hasPP(q_i), mismatch_{q_i}.clear
19 extraFilteringPass(getRoot(Q)) // see Algorithm 17
20 enumerateResults()
21 Procedure processLocalDisjoint(Label q_act,Query node q):
22 |  // pop any element in S_q which is not the ancestor of v_act and update their end
   |  |  values
23 |  while ¬empty(S_q) ∧ getEnd(top(S_q)) < getStart(v_act) do
24 |  |  pop(S_q)
25 Procedure setPositional(Label v_act,Query node q):
26 |  if hasPP(q) then
27 |  |  key = getParentID(v_act)
28 |  |  if counter_q.contains(key) then
29 |  |  |  v_act.position = counter_q.get(key)
30 |  |  |  temp = position++ // the corresponding counter is incremented according
   |  |  |  |  to sharing parents
31 |  |  |  counter_q.replace(key,temp)
32 |  |  else
33 |  |  |  v_act.position = 1
34 |  |  |  counter_q.put(key, 1)
35 |  |  if mismatch_q.contains(key) then
36 |  |  |  v_act.preMiss = mismatch_q.get(key)
37 |  |  |  // preMiss indicates the number of previous siblings which do not satisfy
   |  |  |  |  the pre-structural constraints
38 |  |  else
39 |  |  |  mismatch_q.put(key, 0)
40 Function checkPosition(Label v_act,Query node q):
41 |  if v_act.position satisfies the relational operator in getPPOperator(q) with
   |  |  getPPValue(q) then
   |  |  return : true
42 |  else
   |  |  return : false
```

Algorithm 17: extraFilteringPass(q)

Input: q is a query node
Result: perform a strict filtering as specified by the ordered TPQ the ordering constraints among potential elements

```
1  Procedure extraFilteringPass(Query node q):
2      // read inner query nodes in postorder and any element e which does not satisfy
         the postorder filtering in checkChildMatch(e,q) will be deleted. After that, the
         intermediate list will be resized. Interval pointers for parent query nodes will be
         updated according to the change in their child lists.
3  Function checkChildMatch(Element e, Query node q):
4      key = getParentID(e)
5      temp = 0
6      checkPosition = false; if ¬ checkPreChildMatch(e,q) then
7          if mismatch_q.contains(key) then
8              temp = mismatch_q.get(key)
9              temp++
10             mismatch_q.replace(key,temp)
11         else
12             mismatch_q.put(key,1)
13     if checkPreChildMatch(e,q) ∧ preChildren(q) ≠ ∅ then
14         if mismatch_q.contains(key) then
15             temp = mismatch_q.get(key)
16         // the new value to test the position of the current element against the
             positional value is computed as the previous mismatching siblings plus the
             current mismatching siblings in the extra filtering pass
17         newPPValue = getPPValue(q) + (e.preMiss + temp)
18         if e.position does not satisfy the relational operator in getPPOperator(q)
             with newPPValue then
19             checkPosition = true // it can be safely deleted
       return : checkPreChildMatch(e,q) ∧ checkPostChildMatch(e,q) ∧
                checkChildMatchAll(e,q) ∧¬ checkPosition
20 Function checkPreChildMatch(Element e, Query node q):
21     for n_i ∈ preChildren(q) do
22         if e.start_{n_i} >e.start_{n_i} then
               return : false
       return : true
23 Function checkPostChildMatch(Element e, Query node q):
24     for n_i ∈ postChildren(q) do
25         if e.start_{n_i} >e.start_{n_i} then
               return : false
       return : true
26 Function checkChildMatchAll(Element e, Query node q):
27     for n_i ∈ children(q) do
28         if e.start_{n_i} >e.start_{n_i} then
               return : false
       return : true
```

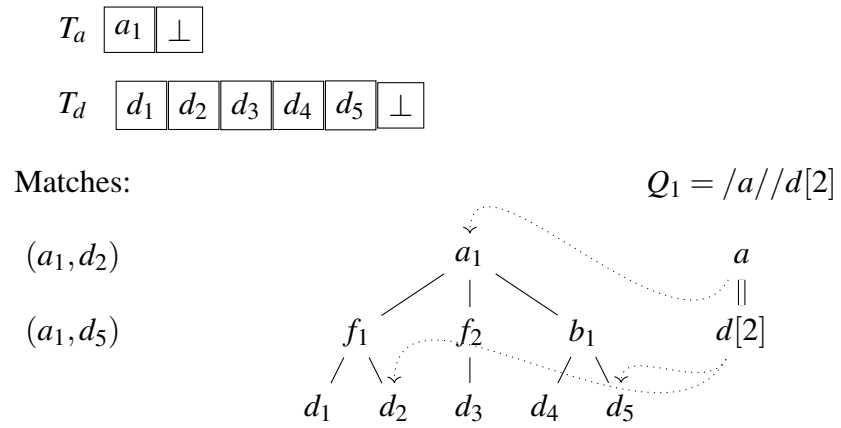


(a) an XML tree T_6. (b) a twig query Q_1

Figure 9.8: An example of maintaining the number of mismatching siblings in preorder and postorder filtering.

the correctness of OPTwigPrime algorithm, it is important to compare it with TwigPos [70] which does not perform preorder filtering. The difference between the algorithms is presented in the following example.

Example 9.6. *Consider the XML tree T_3 in Figure 9.6 and the TPQ Q_2 //a[/y][2]/x, at the beginning of query processing the head elements are $C_{q_a} \rightarrow a_1$, $C_{q_y} \rightarrow y_1$ and $C_{q_x} \rightarrow x_1$. The first call of getMatch() inside OPTwigPrime will return $a \rightarrow a_1$ because it has the positional child and descendant extension. That is, it satisfies the CPL relationship with y. Unlike the original CPL approach, the CPL relationship with x is not considered here because x is connected to a by post-structural constraint. Since a_1 is the first element of its siblings, it is assigned 1 as its positional value and 0 as the number of previously mismatching siblings. After that, a new entry using the parentID value as the hash key is inserted to the counter table with the value set to 1. C_a is forwarded to point to a_2, and a_2 is discarded because it does not have the positional child and descendant extension. When a_2 is advanced, it increases the number of siblings stored in the counter table and adds a new entry to the mismatch table using the parentID as the key which sets the number of mismatching siblings to 1. This is because it fails to satisfy the pre-structural constraint. The next call returns $x \rightarrow x_1$ since it has the smallest start value among elements in streams and it has its ancestor a_1 stored inside the main algorithm. Similarly, $y \rightarrow y_1$ is returned because it violates the positional child and descendant extension of the current element of $C_{q_a} \rightarrow a_3$. Even though x_2 has the smallest start value among elements in streams, it is discarded because it does not have an ancestor extension which was previously returned by getMatch. In contrast, TwigPos will append x_2 to the intermediate storage. Now, the head element a_3 is returned because it has the positional child and descendant extension, therefore it is assigned 3 as its positional value while the attribute preMiss is set to 1 (i.e., a_2 does not satisfy the positional child and descendant extension). y_2 is returned and stored in the inter mediate storage whereas x_3 is discarded because it fails to satisfy the strict prefix matching check with a_3. Finally, elements in the subtree rooted from a_4 are returned and appended to the intermediate storage because they satisfy the positional child and descendant extension. When a_4 is returned, it is has a positional value set to*

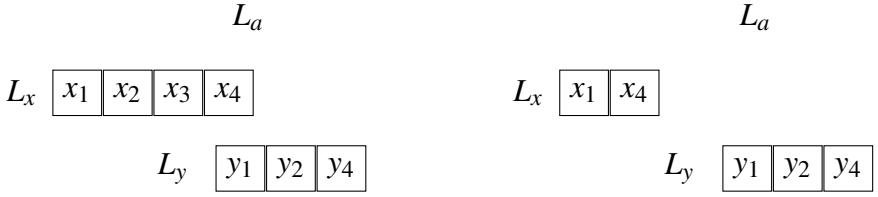


(a) an XML tree T_7.

Figure 9.9: An example of processing a//d with positional predicate.

4 and the preMiss value is set to 1. After all input streams reach the end, OPTwigPrime performs a strict subtree filtering check through extra filtering pass over the intermediate list of a. a_1 *is removed because it does not satisfy the positional predicate. That is, it has positional value which is not equal to the numerical value of the positional predicate plus the number of previously mismatching siblings as* $1 \neq 2+0$. a_3 *is found to satisfy the pre-structural and positional constraints as* $3 = 2+1$. *However* a_3 *is deleted because it does not satisfy the post-structural constraint. Although* a_4 *satisfies both the pre-structural and post-structural constraints, it is removed from the list because the positional predicate is not satisfied as* $4 \neq 2+1$. *Figure 9.10 shows the number of elements stored by each algorithm. The benefits can be seen of combining different filtering strategies to process TPQs with positional predicates.*

Example 9.5 demonstrates the effect of OPTwigPrime in filtering useless elements while storing elements in preorder. The subsection below presents definitions and theorems to prove the correctness of the algorithms proposed. It also analyses their complexities.



(a) The intermediate results of TwigPos

(b) The intermediate results of OPTwigPrime

Figure 9.10: Illustration to the difference between TwigPos [70] and OPTwigPrime.

9.3.3 Analysis of Ordered and Positional Twig Matching Algorithms

This section shows the correctness of the new algorithms and analyses their complexity. The correctness of OTwigPrimeList follows from the correctness of preorder filtering for ordering constraints used in OTJPrimeList described in Section 7.3.3, the correctness of TJStrictPrePrime and TwigPrimeMatch introduced in Chapter 8, with the enumeration algorithm extended to consider ordering constraints or sequence operators during enumerating the output as specified by OTPQs. The correctness holds for TPQs with ordering constraints in addition to both Ancestor-Descendant and Parent-Child relationships. Moreover, the *getMatch* function used by OPTwigPrime is extension to that of OTJPrime which supports A-D and P-C relationships, and assures that the head elements of query nodes with ordering constraints must have start values less than their following elements with the sole exception of query nodes with positional predicates specified as following nodes. The new *getMatch* function takes into account the pre-structural constraints during preorder filtering.

Definition 9.7 (Ordered-Positional Child and Descendant Extension)**.** *A query node q has the ordered-positional child and descendant extension if the following properties hold:*

- *if q does not have pre-structural constraints, then e_q has the child and descendant extension introduced in Chapter 6.*
- *if q has pre-structural constraints, then e_q has the positional child and descendant extension introduced in Definition 9.3.*
- *if $\neg isRoot(q)$, then e_q has a relevant ancestor e_p stored in the main algorithm which has been the head element of a query node p = parent(q) in previous calls of getMatch.*
- *if q has child nodes with ordering constraints, then each of its child elements with the ordering constraints has the ordered and positional extension introduced in Definition 9.4.*

The above definition is essential to establish the correctness of the following lemmas:

Lemma 9.8. *For any arbitrary query node q' which is returned by getMatch(q), the following properties hold:*

1. *q' has the ordered-positional child and descendant extension.*
2. *Either (a) $q = q'$ or (b) q' violates the ordered-positional child and descendant extension of the head element e_q of its p = parent(q') and the head element of $e_{q'}$ has an already processed ancestor in S_p denoted by e_{S_p} for which getEnd(e_{S_p}) > getStart(q').*

Proof. (Induction on the number of ordered-positional children and descendants of q). If q is a leaf query node, it is returned in Line 3 because it satisfies all the properties 1 and 2 in

Lemma 9.8. Otherwise, the algorithm sets up a loop for the inner nodes Lines 4-40 which does not terminate the *getMatch* call until an element is possibly the root of a query match of q. The algorithm recursively gets g_i = *getNext*(n_i) for each child of q in Line 7 if and only if the streams's cursors of *subtree(q)* are not moved from the last call to avoid redundant computations. If for some i, there is $g_i \neq n_i$, and it is known by inductive hypothesis that g_i verifies the properties 1 and 2b with respect to q, so the algorithm returns g_i in Line 9. Otherwise, by the inductive hypothesis that all q's child nodes satisfy properties 1 and 2a with their corresponding sub-queries. If *preChildrenPC(q)* is empty, then at *getElement(q)* (Lines 25-32), *getMatch* advances from T_q all segments that do not satisfy the divisibility by the product of prime numbers in $childrenPC(q)$ returned from *getQCPL*. Otherwise *getMatch* advances from T_q all segments that do not satisfy the divisibility by the product of prime numbers in $preChildrenPC(q)$ returned from *getQCPL*. After that, Lines 10-15 consider the existence of pre-structural constraints to advance from T_q all segments that are beyond the maximum start value of $n_i \in children(q)/n_i \in preChildren(q)$. Then, if q does not satisfy properties 1 and 2a, Lines 28-34 guarantee that $n_i \in children(q)$ with the smallest start value satisfies properties 1 and 2b with respect to the start value of q's head element e_q is returned. Line 28 ensures that if q has post-structural constraints, the element with the smallest start value must be returned. Otherwise, q is checked to see whether or not it has child query nodes with ordering constraints at Line 36. If q is found to be an ordered, branching query node, the algorithm called the subroutine *orderedExtensionWPP* to compare start values for each child of q with ordering constraints against their following elements according to Definition 9.4. If for some j, there is $m_j \prec n_i \wedge \neg preChildren(m_j)$, and it is known by inductive hypothesis that m_j verifies the properties 1 and 2b with respect to q, hence the algorithm returns m_j in Line 38. Otherwise, it is known by inductive hypothesis that all q's child nodes satisfy properties 1 and 2a with their corresponding sub-queries and following elements. After that q is returned at Line 40. □

Lemma 9.9. *Let $e_1, e_2, \ldots, e_m$ be a sequence of elements corresponding to the same query node q and returned by getMatch. Then $getStart(e_1) < getStart(e_2) < \cdots < getStart(e_m)$*

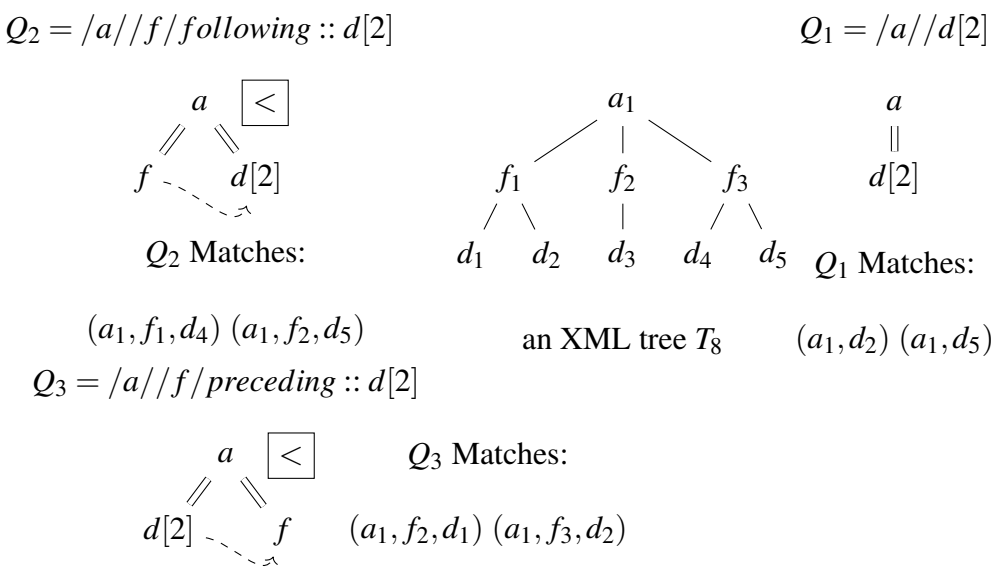
Proof. Query node q is either leaf or internal. If q is a leaf query node, by Lemma 9.8, q satisfies properties 1 and 2b since the parent of q denoted by p=parent(q) has start value greater than the start value of q, such that getStart(q) < getStart(p), *getMatch* returns q at either Line 28 or 34, therefore all elements in the stream of q are returned in ascending order of their start values as they are sorted in their corresponding streams by the definition of *tag streaming scheme*. Otherwise, all elements which are skipped at Lines 16-17 of *getMatch* or Lines 27-28 of *getElement(q)* Lines 7-17 are guaranteed not to be part of any ordered-positional child and descendant extension. By Lemma 9.8, q is returned, so it satisfies properties 1 and 2a. Thus all elements in the stream of q are returned in ascending order of their start values as they are sorted in their corresponding stream *tag streaming scheme*. For both cases the lemma holds. □

The above lemmas guarantee that all elements in the XML tree which are part of some solution at a subtree rooted at a query node in a TPQ will be returned in document order, and each time *getMatch* returns a query node q and the head element of q must have the ancestor extension. Using the above lemmas (Lemma 9.8 and 9.9) and lemmas introduced in Sections 6.4.2, 7.3.3 and 9.3.3, the next theorem will be used to prove the correctness of OPTwigPrime and its core function *getMatch*.

Theorem 9.10. *Given a twig pattern query Q and an XML document D, Algorithm OPTwigPrime correctly construct the intermediate results of Q on D.*

Proof. In Algorithm *OPTwigPrime*, *getMatch(root)* is repeatedly invoked to determine the next query node to be processed. Using Lemma 9.8, it is known that all elements returned by $q_{act} = getMatch(root)$ have the ordered-positional child and descendant extension. If $q_{act} \neq root$, Line 7, the algorithm sets the end values for all elements in the intermediate lists $L_{parent(q_{act})}$ that are not ancestors of the head element of q_{act} by Lemma 6.21. After that, it is already known q_{act} has an ordered-positional child and descendant extension and it has the ancestor extension so that this guarantees its participation in a weak match of prefixed path from itself to the root. Then, $S_{q_{act}}$ is cleaned by popping elements which do not contain the current element v_{act} so that they update end values of their intervals. If q_{act} has a positional predicate, v_{act} is assigned its positional value using the *parentID* as the key. Lines 10-14 ensures that the current element has a strictly matched prefixed path. If the head element fails to pass a strict prefix path filtering, then it can be skipped safely to proceed to the next cycle at Line 15. Otherwise, the start positions of intervals for v_{act} are recorded and v_{act} is pushed into $S_{q_{act}}$ and appended to its corresponding intermediate list, using Lemma 8.5 unless q_{act} has a positional predicate but does not have pre-structural constraints, it must satisfy the positional predicate to be considered for further processing via the supporting function *checkPosition*(q_{act}). Finally, the stream of q_{act} is advanced and the algorithm proceeds to the next iteration. When all streams are ended, the procedure extraFilteringPass is called to provide a strict subtree filtering match and filter out elements which do not satisfy positional predicates. Once the intermediate lists containing elements with their intervals are correctly set, it is straightforward to enumerate the output. □

The correctness holds for TPQs which may contain positional predicates and ordering constraints, and pre-structural constraints consisting of both Ancestor-Descendant and Parent-Child relationships whereas pre-structural constraints can contain ordered axes and sequence operators, along with both Ancestor-Descendant and Parent-Child relationships. More importantly positional predicates are only on edges corresponding to A-D and P-C axes in TPQs. The reason for this is that the new approach can not guarantee optimal enumeration when ordered axes and sequence operators are present in order to update the number of mismatching siblings during the filtering phase. Therefore, *time point 2* for computing positional values for query nodes in the third and fourth case should be performed during the output enumeration. It should be noted that when positional



(a) an XML tree T_8 and three TPQs with ordered axes and positional predicates, Q_1, Q_2 and Q_3 .

Figure 9.11: An example to illustrate the difference between positional predicates on basic axes as in Q_1 and ordered axes as in Q_2.

predicates are on edges corresponding to ordered axes in the original TPQ, the *counter* and *mismatch* tables should be modified to use the information of the current element in the intermediate storage as the key according to the XPath specification. If edges are backward axes, positional predicates must be checked in *reverse document order*. Otherwise, elements must satisfy positional predicates in *document order*. For illustration, consider the XML tree T_8 and two TPQs with positional predicates Q_1 and Q_2 in Figure 9.11. For Q_1, the positional information of the context element is determined using the information of the parent whereas the positional information of the potential element for Q_2 is based on its location according to the preceding element. Though d_4 is useless for Q_1, it is useful for Q_2. On the other hand, Q_3 returns matches which satisfy the positional predicate in postorder as d_1 and d_2. While it is straightforward to process positional predicates on forward axes since the intermediate results are stored in preorder, positional predicates on backward axes can be supported by reading the parents' intervals backwards to set positional values correctly. That is, elements within child intervals are scanned starting from end values and countdown to start positions.

The new algorithms read elements form data streams only once in a single forward scan through advanced preorder filtering functions. When elements are appended to the intermediate storage, each child check and interval set takes constant time. Therefore, the worst-case time complexity for building up the intermediate storage is $O(D \times |Input|)$ where D is the maximum degree at any query node in TPQ with n query nodes and Input is the sum of the lengths of the n input lists. When the buffering technique is used to process OTPQS, the worst case behaviour for the algorithms is $O(|P| \times |F| + D \times |Input|)$ where P is the sum of lengths of input lists for query nodes with ordering constraints and F is the sum of lengths of input lists for query nodes pointed to by ordering constraints, while

Input is the sum of the lengths of the remaining input lists without ordering constraints. Therefore, the intermediate result can be enumerated in linear time $O(|Output|)$ where Output is the number of twig matchings.

Similar to TJStrictPrePrime and TwigPrimeMatch, the worst case space complexity of OPTwigPrime algorithms is $O(|Input|)$ which is linear with respect to the total number of elements whose tags appear in TPQs. This is due to the fact that it directly constructs the intermediate results. The maximum number of entries for the *hash* tables is $O(2 \times N \times P)$ where N is the number of nodes with positional predicates in TPQs and P is the total number of parents in the XML tree. By contrast, the worst case space complexity of the OTwigPrimeList algorithms is equal to the sum of lengths of input lists corresponding to query nodes with the ordering constraints plus the total number of elements whose tags appear in TPQs as $O(|Output| + |Input|)$.

The next section describes the experiments to test the performance of bottom-up holistic twig matching algorithms for processing ordered axes, sequence operators and positional predicates proposed in this thesis.

9.4 Experimental Evaluation

The following experiments investigate the effectiveness of the new approaches by performing a number of experiments on benchmark, real world and synthetic datasets. The effect of the ordered child and descendant extension and the use of buffering techniques to process ordered TPQs in bottom-up holistic approaches will be explored. In addition, the experiments investigate the benefits of early identification of elements that satisfy positional predicates. This section provides the experimental results of a comparison of the performance and scalability of the new approaches with the existing bottom-up holistic twig matching algorithm, TwigPos which provides early filtering of *following-sibling* relationships and positional predicates. Two parallel experiments will be performed with the XMark, TreeBank and Random datasets, each aimed at distinctive subclass of TPQs: *experiment 1* tests all the three related algorithms on OTPQs, namely OTwigPrimeList, OPTwigPrime and TwigPos proposed in [70]. Note that TwigPos is implemented with a straightforward postprocessing in the enumeration phase to support the rest of the ordering constraints in order to compare the performance with the proposed algorithms. *Experiment 2* tests OPTwigPrime and TwigPos, which is extended to support positional predicates under A-D edges, on TPQs with positional predicates and ordering constraints specified as post-structural while pre-structural constraints consist of both A-D and P-C axes. It should be noted that no performance comparison with the EPPP (Effective Positional Predicate Processing) structural join algorithm proposed in [217] is presented because it does not fit into the holistic general framework. Also, TwigPos is previously reported in [70] to have better performance than EPPP for queries similar to that used in *experiment 2*. Eventually,

Table 9.1: Experimental TPQs with positional predicates for the XMark dataset.

Code	XPath expression	Result size
PXQ_1	//people/person[//address/zipcode][88]/profile/education	1
PXQ_2	//text[/bold][//keyword[10]]//emph	100
PXQ_3	//description[//text]//parlist[1]	98602
PXQ_4	//mail/text[/keyword]//bold[3]	1632
PXQ_5	//description/text/bold[5]	196
PXQ_6	//item[/name[1]]//description[//text/keyword[6]]	90
PXQ_7	//item/name[1]/following-sibling::payment/following-sibling::description//text/keyword[6]	90
PXQ_8	//people/person[88]/profile/education	1

Table 9.2: Experimental TPQs with positional predicates for the TreeBank dataset.

Code	XPath expression	Result size
PTQ_1	//S/VP/PP[/NP/VBN][2][/IN]	3
PTQ_2	//NP/VP/_NONE_[2]	6
PTQ_3	//NP[/NN[6]]/PP	1
PTQ_4	//S//VP//PP[//IN][4]/NP	65
PTQ_5	//WHNP[/NP//JJ][position() $\geq$ 1]/WRB	1
PTQ_6	//S/VP[2]/PP[/IN]/NP[/CD]/VBN	1
PTQ_7	//S/VP/PP/IN/following-sibling::NP/VBN[2]	1
PTQ_8	//NP/PP[3]/following-sibling::NN	1

all the algorithms tested in the experiments were implemented and added to the query processor described in Section 5.2.2.2.

For *experiment 1*, the ordered queries on the XMark, TreeBank and Random datasets are similar to those used in the experiments of Chapter 7. For *experiment 2*, as there are no standard TPQs with positional predicates available for this experiment, test queries are designed with balanced consideration of the following factors: depth, width, skewed, balanced, and mixture of pre-structural and post-structural constraints. It should be noted that some of TPQs with positional predicates on XMark and TreeBank datasets were obtained from [70]. However, TPQs with positional predicates over the Random dataset were designed based on the factors mentioned above. For the sake of simplicity, TPQs with positional predicates were encoded so that the code indicates the dataset and the corresponding TPQ. By way of example, PRQ_2 refers to the second positional TPQ issued over the Random dataset. The characteristics of the test queries over XMark are presented in Table 9.1. The properties of TPQs selected over TreeBank are given in Table 9.2. Table 9.3 provides an overview of TPQs over the Random dataset. Both *experiment 1* and *experiment 2* compare the algorithms based on metrics similar to those described in Section 8.4.2. The scalability tests are provided in Section 9.4.3.

Table 9.3: Experimental TPQs with positional predicates for the Random dataset.

Code	XPath expression	Result size
PRQ_1	//b/e//a[/f][2]//d	16
PRQ_2	//a//b[//e]/c[3]	290
PRQ_3	//a/c//b[//d][2][/e]/f	2
PRQ_4	//b/c[/e][3]/f	4
PRQ_5	//d[a[//e]/f]/c[position $\leq$ 3]/b	0
PRQ_6	//b//e//a[//f][2][d]	23
PRQ_7	//a/c//b[3][/following-sibling::d][/following-sibling::e]	1
PRQ_8	//a//d[2][/following-sibling::f]/following-sibling::e[/c]//b[2]	3

9.4.1 Experiment 1: Ordered Twig Queries

9.4.1.1 Experimental Results

This section presents the evaluation of the experimental results. The first step in this process was to inspect all the results returned from the algorithms. Since the algorithms produced the same results, the correctness of the new approaches can be verified. For the purpose of clarification, the discussion of the query performance related to a particular dataset is presented within an individual subsection. The query performances for OTPQs over XMark, TreeBank and Random datasets are evaluated in Sections 9.4.1.2, 9.4.1.3 and 9.4.1.4.

9.4.1.2 XMark

This section discusses the experimental results for OTPQs over the XMark dataset. The purpose of using this dataset is to show that the new approaches do not bring any overhead for processing XML documents with a simple structure. Figure 9.12 shows the number of elements stored in the intermediate storage by each algorithm along with the actual elements which are part of complete matches. An immediate observation from the figure is that the new approaches are more efficient in terms of the intermediate results than TwigPos for all queries on this dataset. It can be seen from the data in Figure 9.12, OTwigPrimeList provided optimal evaluation for all queries, and the size of the intermediate storage produced by OPTwigPrime and TwigPos is 2 and 12 times larger than the intermediate storage of OTwigPrimeList, respectively. Generally, when comparing OPTwigPrime with TwigPos, OPTwigPrime stored fewer elements than TwigPos. To assess the query performance, the Kruskal-Wallis test was carried out to see whether or not there is a performance difference between two algorithms at least for every OXQ on the dataset. The results of the groups analysis are set out in Table 9.4. As can be seen from the table, a significant difference between two groups (i.e., algorithms) at least was evident in all ordered queries.

In consequence, the number of paired comparisons for this dataset can be obtained using Formula 6.2 described in Chapter 6 as $= \frac{(3\times(3-1))}{2} \times 5 \ = 15$. Details of the pairwise

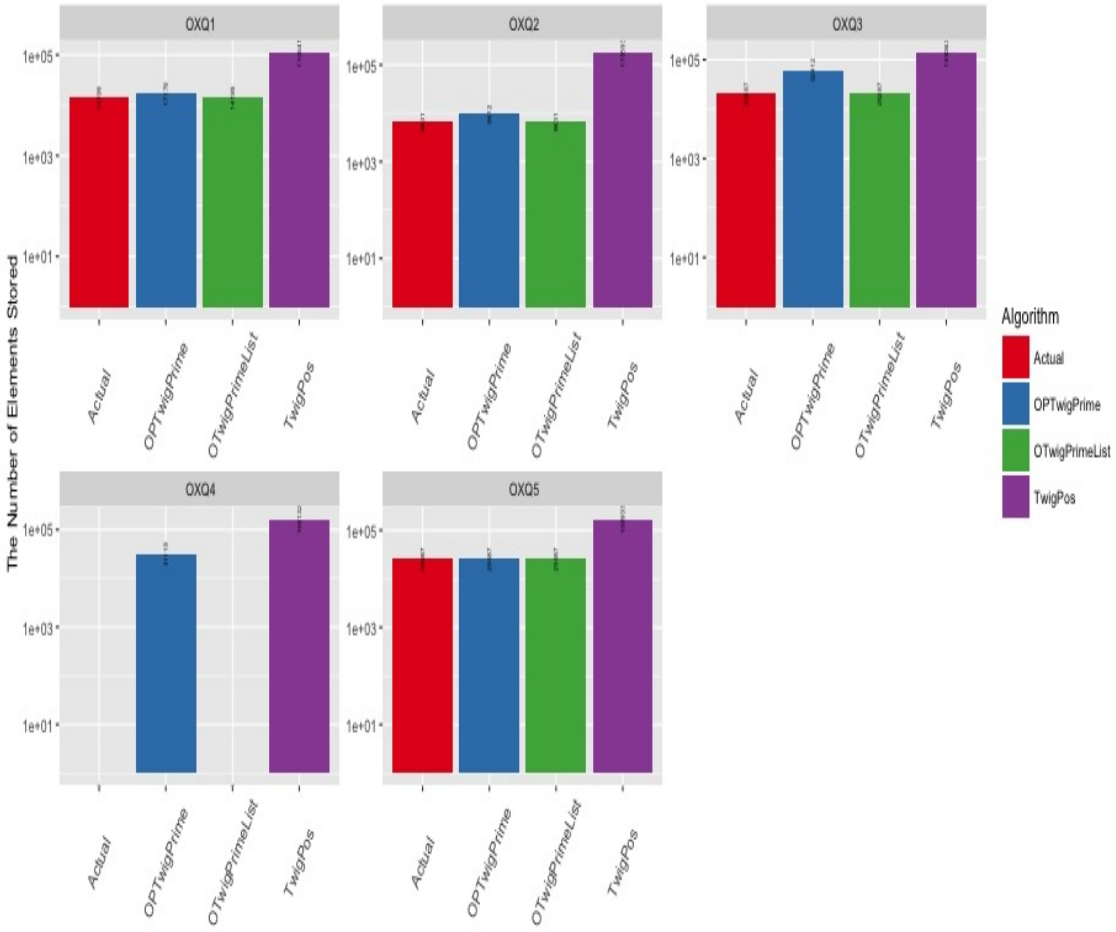


Figure 9.12: The number of elements stored in the intermediate storage by each algorithm for the ordered queries tested over the XMark collection. "Actual" represents the number of elements relevant to the ordered query results.

comparisons can be found in Appendix D. The performance result of ordered queries on this dataset is plotted in Figure 9.13. Clearly, the new algorithms were significantly faster than TwigPos on all OTPQs. According to the results provided in Table 9.5, OPTwigPrime had the best performance in all cases with the sole exception of OXQ_3. OTwigPrimeList significantly outperformed the other algorithms in the experiments for OXQ_3 in which it took only 20% of the time taken by the other algorithms. A possible explanation for this might be that OXQ_3 is a simple OTPQ with only one branching query node so that the buffering technique did not cause any overhead during the filtering phase and the cost of enumerating results because a redundant scan is avoided. Even though OPTwigPrime applied a weak filtering among ordered, sibling query nodes, it had a superior performance to the algorithms compared. TwigPos was comparable to OTwigPrimeList only when a query contained the minimal ordered constraint as for OXQ_5. TwigPos ran slightly faster than OTwigPrimeList. For each pairwise comparison, the effect size suggested that there is a medium to large practical significance.

On the whole, the experimental results demonstrated that the novel approaches to match ordered TPQs had a superior performance to the existing algorithm, TwigPos, which

Table 9.4: Results for the comparison groups on the XMark dataset.

Query	p-value	p-value < 0.05
OXQ1	1.904992e-58	TRUE
OXQ2	3.411924e-52	TRUE
OXQ3	1.340419e-56	TRUE
OXQ4	1.899373e-58	TRUE
OXQ5	4.434081e-49	TRUE

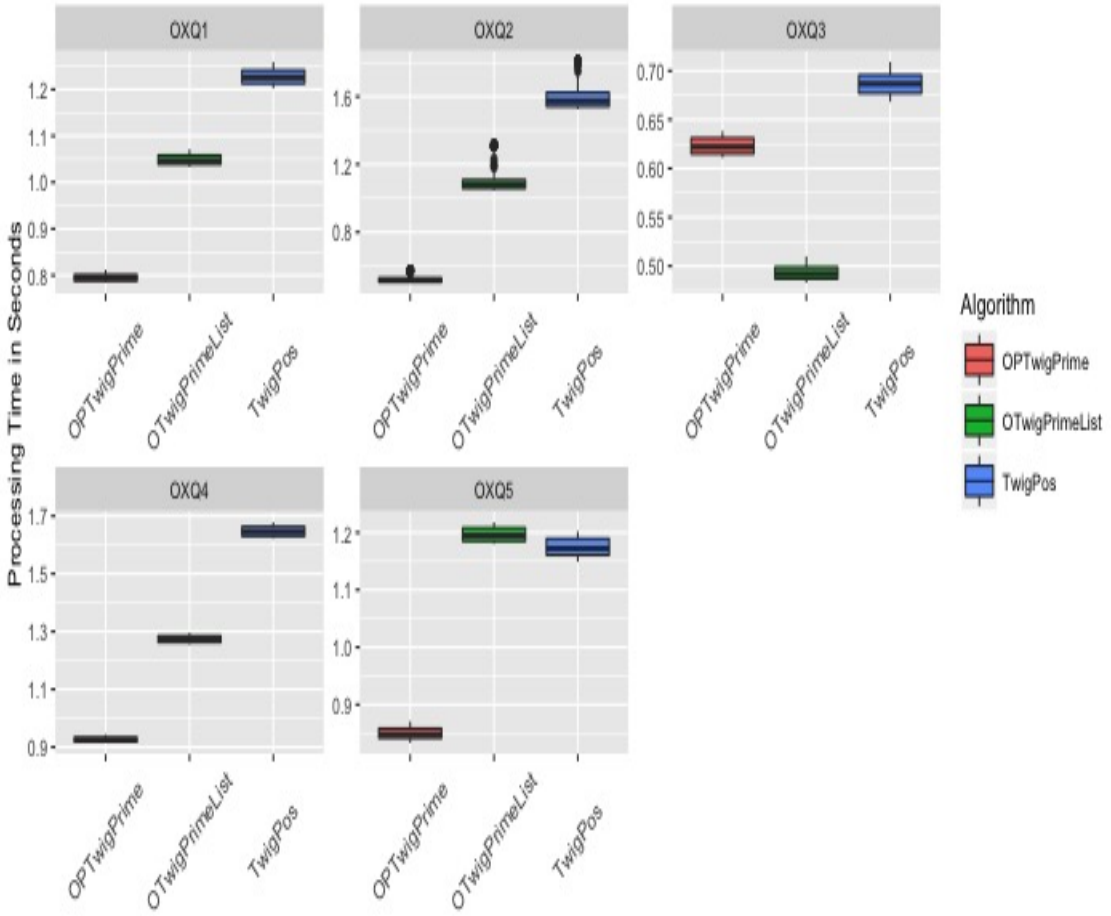


Figure 9.13: Query processing time of the algorithms compared for OTPQs against XMark.

Table 9.5: The overall comparisons based on U tests over the XMark dataset. "-" indicates no statistically difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
OTwigPrimeList	5	5	0
OPTwigPrime	9	1	0
TwigPos	1	9	0

supports only *following-sibling* axes and uses postprocessing for the remaining of ordering constraints, in terms of the space consumption and query running time. For the XMark dataset, the extra filtering strategy applied by the buffering algorithm, OTwigPrimeList provided optimal evaluation for all ordered queries. However, the buffering technique had

Table 9.6: Results for the comparison groups on the TreeBank dataset.

Query	p-value	p-value < 0.05
OTQ1	5.612397e-58	TRUE
OTQ2	2.858499e-38	TRUE
OTQ3	2.735609e-54	TRUE
OTQ4	3.833918e-49	TRUE
OTQ5	6.147709e-58	TRUE
OTQ6	3.514137e-58	TRUE
OTQ7	1.173357e-56	TRUE
OTQ8	4.059079e-55	TRUE
OTQ9	3.094025e-58	TRUE

no effect and only caused overhead with respect to the query performance when compared with the simple algorithm, OPTwigPrime. To summarize, OPTwigPrime significantly outperformed in all ordered queries the other algorithms with the exception of OXQ_3 which was slower than OTwigPrimeList as can be seen in Table 9.5.

9.4.1.3 TreeBank

The experiment is designed to test algorithms processing ordered TPQs on the TreeBank dataset which have different structures and a mixture of A-D, P-C, LR, SLR and SeqLR edges. Figure 9.14 shows the size of the intermediate storage constructed by each algorithm. It can be seen that by far the best performance is achieved by the buffering algorithm, OTwigPrime. However, it stored irrelevant elements in ordered TPQs with LR ordering constraints, namely OTQ_1, OTQ_5, OTQ_7 and OTQ_8. These results are consistent with the theoretical analysis discussed in Chapter 7. That is, the buffering lists may contain elements with ordered axes or sequence operators which are found to satisfy the LR or SeqLR ordering relationships with elements in subtrees rooted from ancestors which are not their *lowest common ancestor* within a deeply recursive XML tree. Another reason is that the buffering lists are checked in one traversal starting from preceding to following elements so that some following elements may be found useless later if they also precede another elements. Interestingly, there was not much difference between the buffering technique and the simple ordered filtering checks in spite the fact that the TreeBank dataset has many recursive tags. This may be due to the nature of the TreeBank document in which elements are ordered in the sequence they occur in the original text so that the use of the minimal ordering constraint can eliminate irrelevant elements efficiently. On the other hand, TwigPos stored elements 11 and 18 times larger than OPTwigPrime and OTwigPrimeList, respectively. TwigPos provided optimal enumeration for OTQ_2 and OTQ_4 because they only have *following-sibling* relationships. However, the size of the intermediate storage of TwigPos for OTQ_2 and OTQ_4 was about 83% and 411% larger than that generated by OTwigPrimeList.

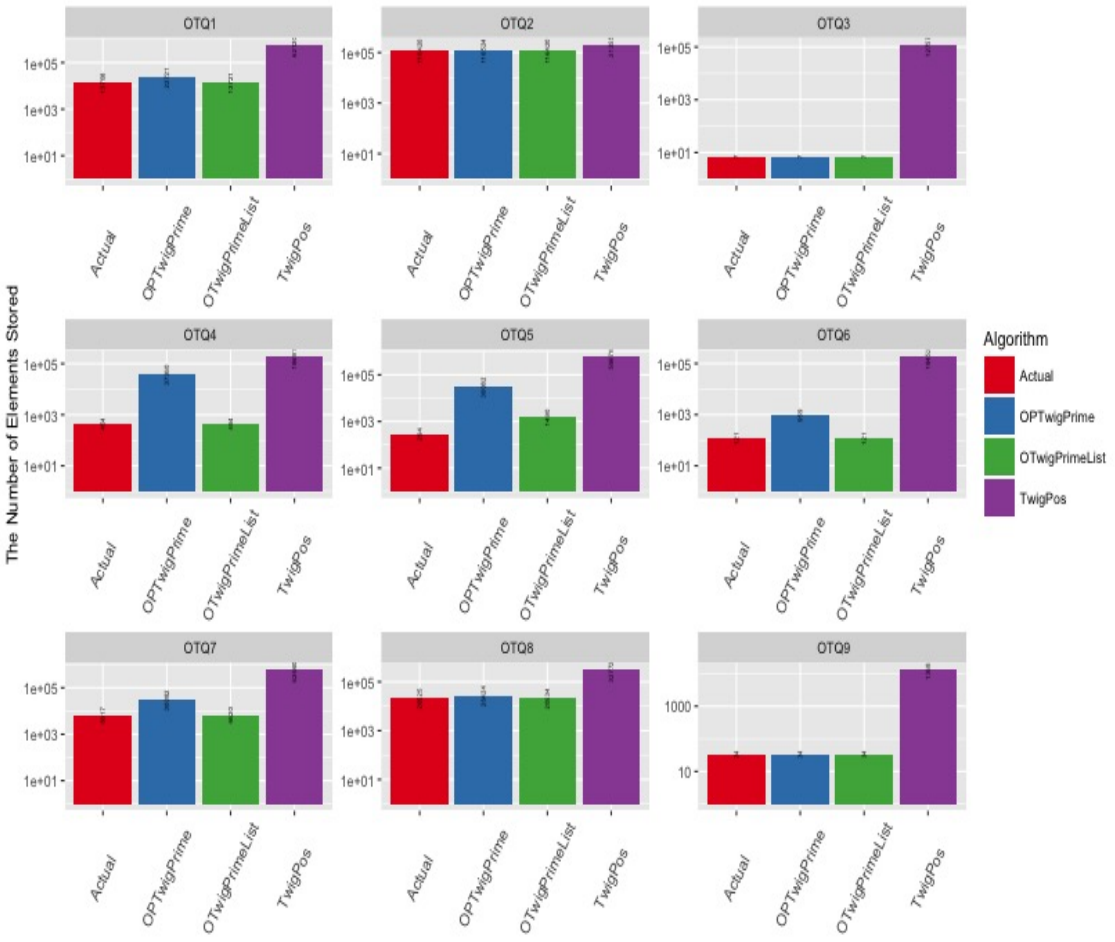


Figure 9.14: The number of elements stored in the intermediate storage by each algorithm for the ordered queries tested over the TreeBank document. "Actual" represents the number of elements relevant to the ordered query results.

Moving now to evaluate the query performance, the Kruskal-Wallis tests presented in Table 9.6 revealed that there is a difference in the performance between at least two algorithms. Accordingly, paired comparisons based on the U test of Mann Whitney were computed. The number of pairwise comparisons for this dataset can be obtained using Formula 6.2 as $= \frac{(3\times(3-1))}{2} \times 9 \ = 27$. The full results of the pairwise comparisons can be found in Appendix D. From the data in Figure 9.15 and Table 9.7, it is apparent that OPTwigPrime significantly outperformed the other approaches in all queries. Note that the cost difference between OPTwigPrime and OTwigPrime which is due to some run-time overhead in order to improve the filtering strategy. The effect of the extra filtering is significant in top-down approaches as was previously reported in Chapter 7, whereas the use of a combination of CPL and level split approach in bottom-approaches can lower the cost of enumerating results without any additional overhead. The reason for this is that the buffering algorithms, OTJPrimeList and OTJPrimeMultiLists can overcome the main weakness of top-down approaches which comes from redundant intermediate results and inefficient output enumeration. TwigPos was comparable to OTwigPrimeList only for

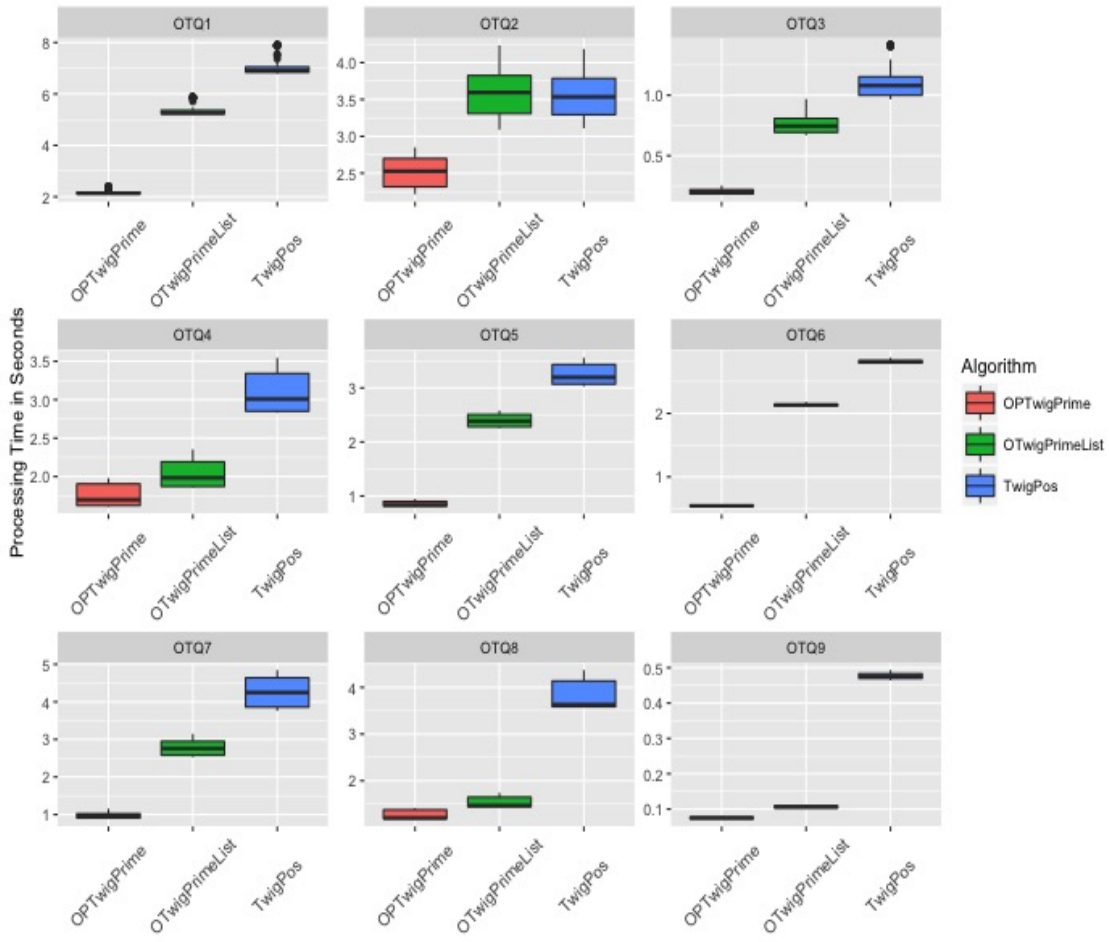


Figure 9.15: Query processing time of the algorithms compared for OTPQs against the TreeBank document.

OTQ_2 because the *following-sibling* axis is present and the number of elements stored by all algorithms is quite large.

Table 9.7: The overall comparisons based on U tests over the TreeBank dataset. "-" indicates no statistically difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
OTwigPrimeList	8	9	1
OPTwigPrime	18	0	0
TwigPos	0	17	1

To sum up, OPTwigPrime which applies a simple ordering check over the head elements of candidate parents showed a superior performance to the other algorithms in terms of query running time on deep recursive data (e.g., TreeBank). On the other hand, OTwigPrimeList stored fewer elements than the other algorithms due to the fact that it uses the buffering technique to filter out useless elements with ordering constraints.

9.4.1.4 Random

The Random dataset has a complex structure with six distinct tags. This dataset was selected to show the differences between algorithms where the XML combines features of data-centric and text-centric documents, being relatively structured and deeply recursive at the same time. The number of elements stored by algorithms in the experiment is plotted in Figure 9.16. Unlike the previous datasets, OTwigPrimeList provided optimal evaluation only for one OTPQ, ORQ_6 because elements in subtrees matching the ordered query satisfy the SeqLR ordering constraint. Thus, OPTwigPrime did not store any useless element for ORQ_6. OTwigPrimeList stored two orders of magnitude fewer elements than TwigPos while the number of elements stored by OPTwigPrime was twice as big as the number of element stored by OTwigPrimeList. Further, the OPTwigPrime algorithm showed a superior performance in filtering out useless elements despite the complexity of this dataset in terms of structure. Generally, the effect of the new approaches is significantly superior to the existing algorithm, TwigPos. The performance results with ordered queries over the Random dataset is plotted in Figure 9.17. To evaluate the query performance, the Kruskal-Wallis test was carried out to test the null hypothesis stating that there is no difference in the performance between the algorithms tested. The results of the groups analysis are presented in Table 9.8. It can be seen from the data in Table 9.8 that every Kruskal-Wallis test revealed that there is a significant difference between two algorithms at least.

Table 9.8: Results for the comparison groups on the Random dataset.

Query	p-value	p-value < 0.05
ORQ1	1.93E-58	TRUE
ORQ2	1.08E-43	TRUE
ORQ3	1.93E-58	TRUE
ORQ4	9.22E-54	TRUE
ORQ5	1.93E-58	TRUE
ORQ6	1.93E-58	TRUE
ORQ7	1.93E-58	TRUE

Accordingly, the total number of paired comparisons for the Random dataset can be computed using Formula 6.2 described in Chapter 6 as $= \frac{(3\times(3-1))}{2} \times 7 \ = 21$. The full results of the pairwise comparisons can be found in Appendix D. The overall results are provided in Table 9.9 which summarises the comparisons to show how many times each algorithm statistically was either faster or slower. As shown in Figure 9.17, the new algorithms were significantly faster than TwigPos on all ORTQs. OPTwigPrime had the best performance in all cases, and it was several times faster than the rest of algorithms tested. It was five times faster than OPTwigPrimeList and fifty times faster than TwigPos. Even though most of the ordered queries over this dataset have *following-sibling* axes, TwigPos failed to outperform the new approaches nor lower the cost of building the

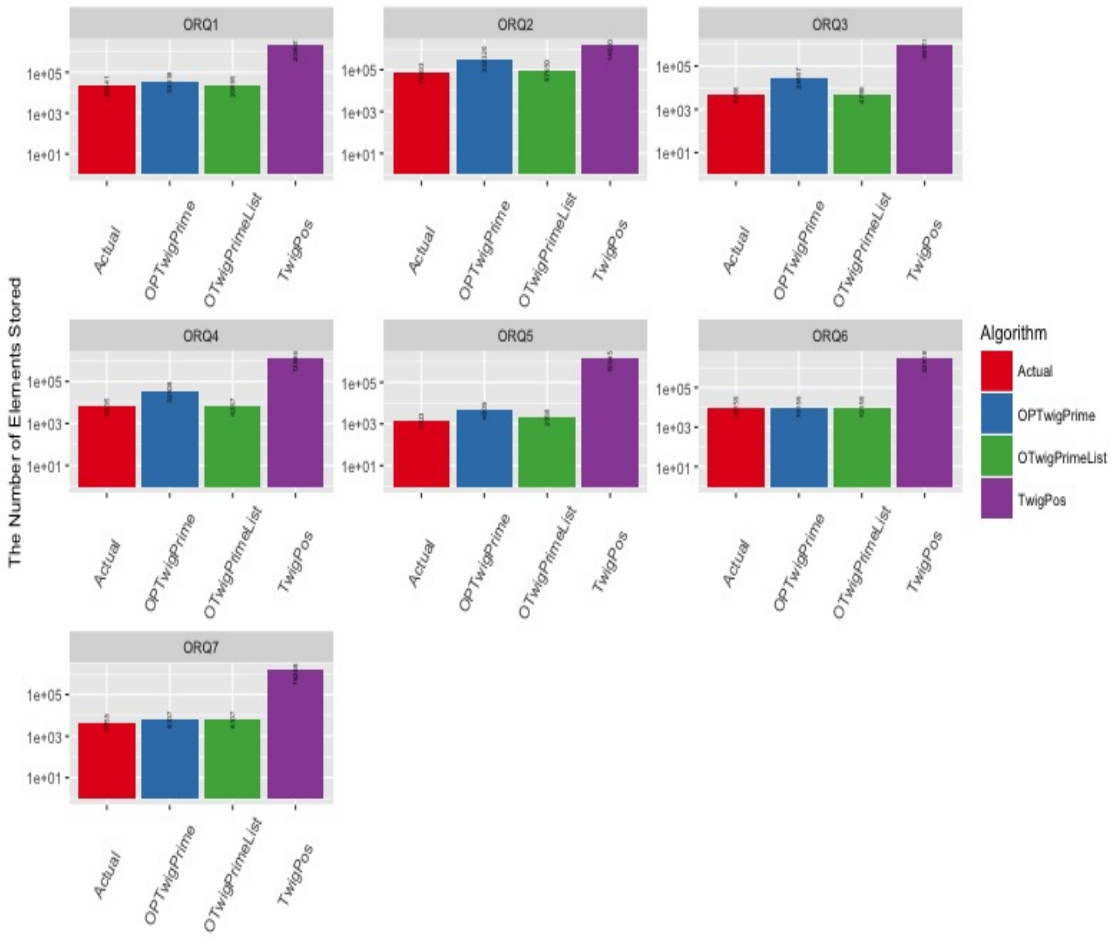


Figure 9.16: The number of elements stored in the intermediate storage by each algorithm for the ordered queries tested over the Random dataset. "Actual" represents the number of elements relevant to the ordered query results.

intermediate storage. When comparing OTwigPrimeList with TwigPos, OTwigPrimeList significantly outperformed TwigPos in all queries as presented in Table 9.9, except for ORQ_2 where OTwigPrimeList moderately outperformed TwigPos. After tracing into the evaluation of this query, it can be observed that all algorithms generated a quite large number of elements, and there is a *following-sibling* edge. For every U test, the effect size suggested that there is a medium to large practical significance except for ORQ_2, where low practical significance was evident when a paired comparison between OTwigPrimeList and TwigPos.

Table 9.9: The overall comparisons based on U tests over the Random dataset. "-" indicates no statistically difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
OTwigPrimeList	7	7	0
OPTwigPrime	14	0	0
TwigPos	0	14	0

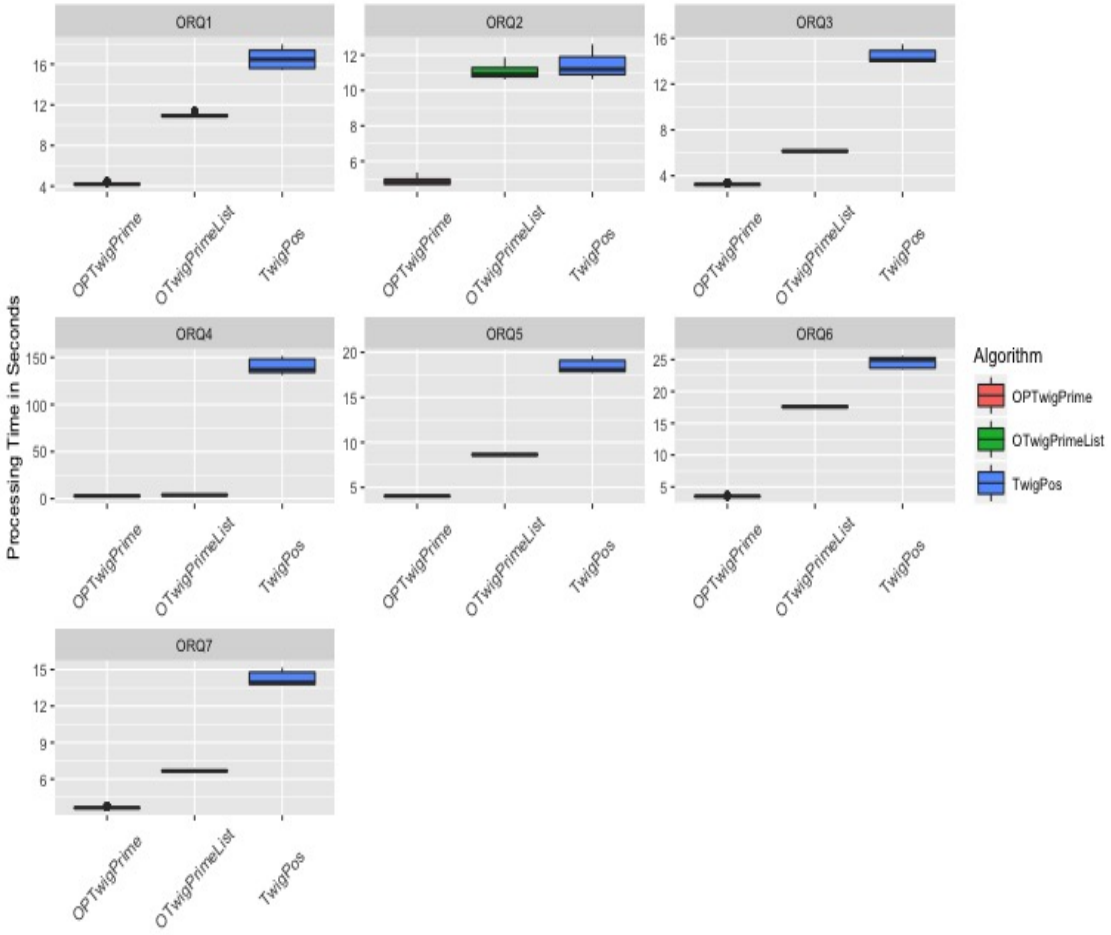


Figure 9.17: Query processing time of the algorithms compared for OTPQs against the TreeBank document.

To conclude, the experiment was used to explore the effects of combining the ordered child and descendant extension introduced in Chapter 7 and level split approach as the main intermediate storage over the Random dataset. The complexity of this dataset comes from the fact that every subtree may form a potential match. Thus, the new approaches provided an efficient way to perform strict filtering checks although ordering constraints are present (see Figure 9.16). Clearly, OPTwigPrime showed a superior performance to the rest of algorithms compared, due to the fact that it uses a simple filtering strategy which does not bring any additional overhead while filtering out a considerable number of useless elements during the filtering phase. In addition, OTwigPrimeList was found to be several times faster than TwigPos. For example, it was 35 times faster than TwigPos for ORQ_4 (see Figure 9.17).

9.4.2 Experiment 2: Ordered/Positional Twig Queries

This section presents the evaluation of the experimental results for processing Ordered/-Positional twig queries. This experiment is limited to OPTwigPrime and TwigPos only. Even though OPTwigPrime is capable of processing TPQs which may contain ordering

constraints as post-structural, the experiment only contains *following-sibling* relationships in the post-structural constraint because TwigPos is capable of evaluating this type of twig. Thus, the experiment aims at comparatively studying the performance of OPTwigPrime and TwigPos on various common subclasses of the aforementioned type of Ordered/Positional twig queries. In order to verify the validity of the new approaches, all query results in the experiments returned from the tested algorithms were inspected. Since both algorithms returned the same results, their correctness can be verified. To allow precise comparisons, the discussion of the query performance related to a particular dataset is contained within an individual subsection. The query performances for TPQs with ordered constraints and positional predicates over XMark, TreeBank and Random datasets are discussed in Sections 9.4.1.2, 9.4.1.3 and 9.4.1.4, respectively.

9.4.2.1 XMark

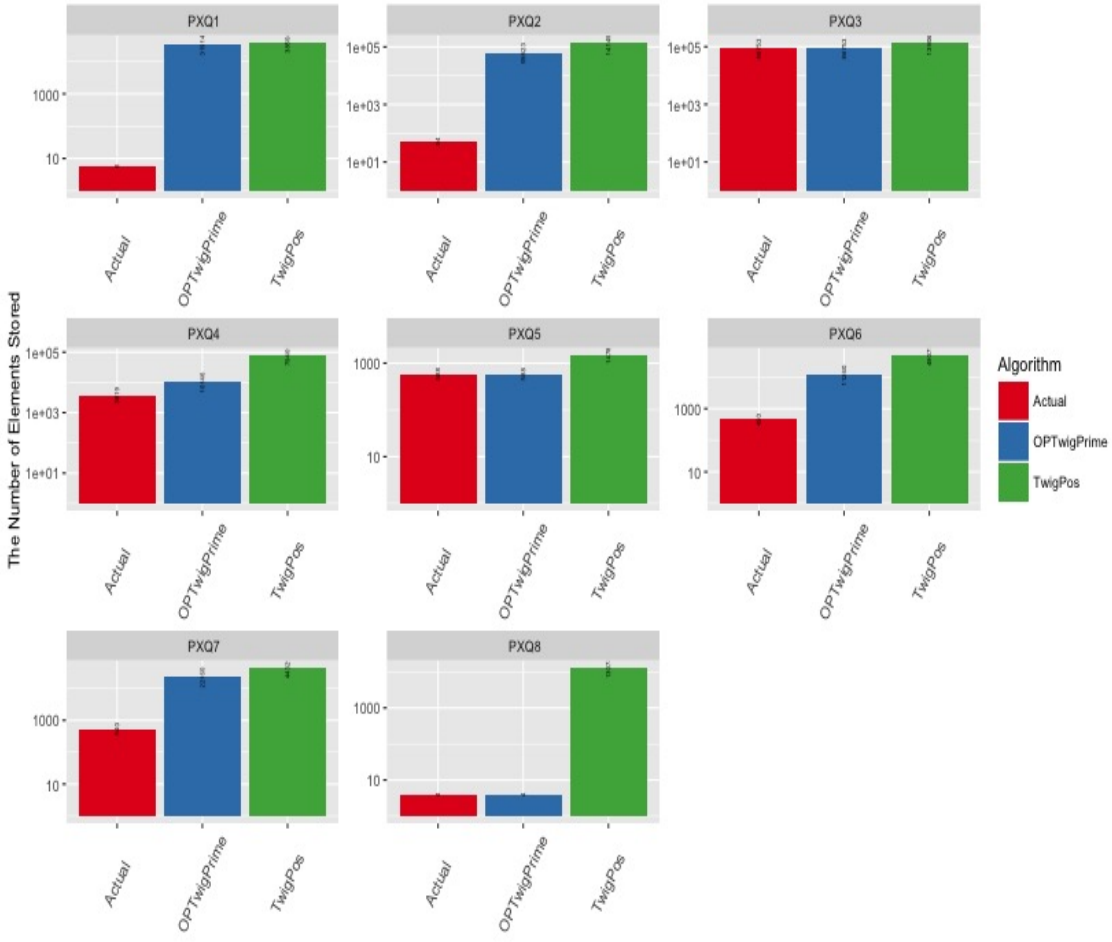


Figure 9.18: The number of elements stored in the intermediate storage by each algorithm for the ordered queries tested over the XMark collection. "Actual" represents the number of elements relevant to the query results.

In the XMark dataset, the experiment is to compare the performance of the algorithms on a relatively balanced XML tree. The number of elements stored by each algorithm are

shown in Figure 9.18. Generally, the size of the intermediate storage of OPTwigPrime was half as big as that constructed by TwigPos due to the fact that it uses a combination of preorder and postorder filtering whereas TwigPos uses only a postorder filtering strategy. Since positional predicates are checked during the postorder filterings, OPTwigPrime stored many irrelevant elements for PXQ_1 and PXQ_2. For example, OPTwigPrime stored 59,823 elements while TwigPos stored 141,454 elements; however only 54 elements are relevant to PXQ_2. Note that PXQ_8 is a path query with positional predicate. However, TwigPos stored a large number of irrelevant elements. For PXQ_8, OPTwigPrime provided optimal evaluation and stored four orders of magnitude fewer elements than TwigPos.

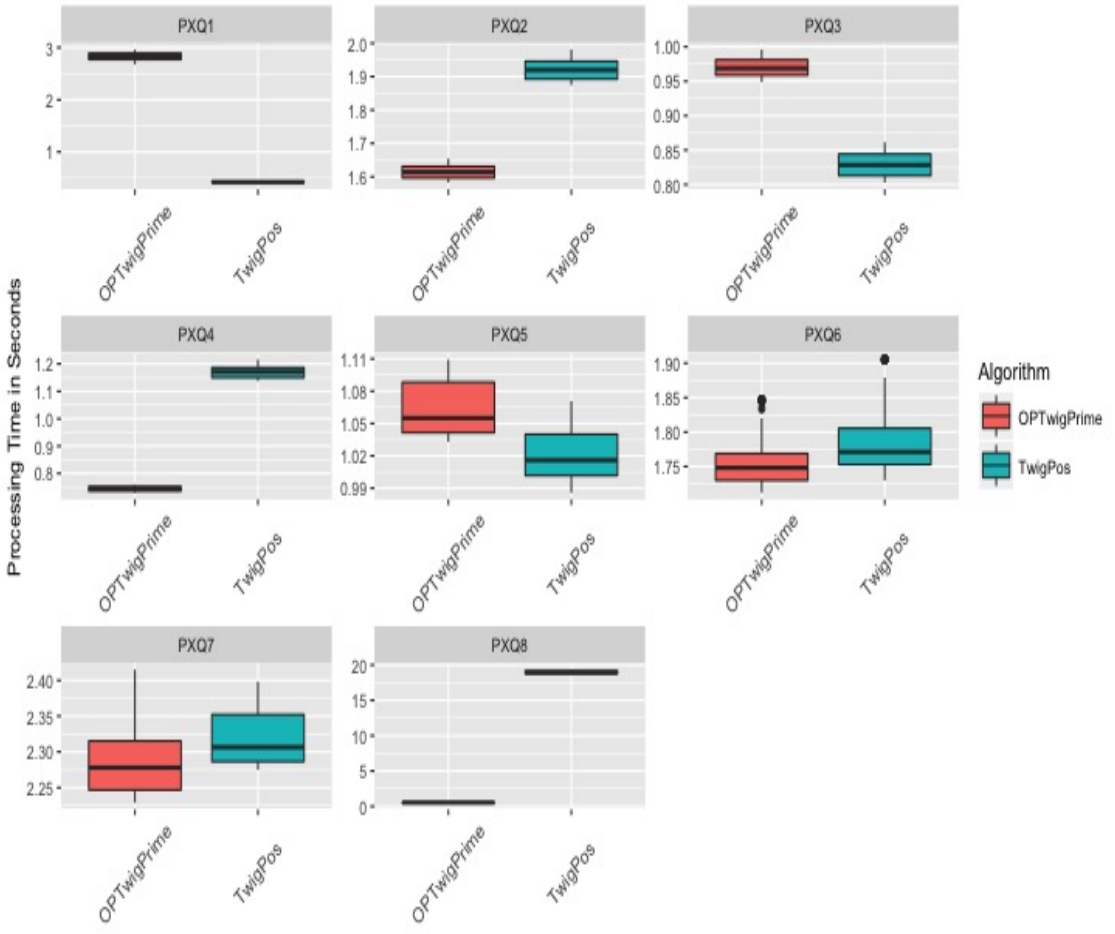


Figure 9.19: Query processing time of the algorithms compared for Ordered/Positional over the XMark document.

Moving on now to compare query processing time for this dataset, Figure 9.19 presents the performance results. To compare the query performance statistically, the experiment is based on hypothesis that there is no difference in the performance between the algorithms so that the Mann-Whitney U test was carried out to test the null hypothesis for each query. Thus, the total number of paired comparisons is equal to the number of queries since the experiment tests two algorithms. All U tests suggest that there is a significant difference in the performance between the algorithms. The summary of the paired comparisons based on the Mann Whitney U test is provided in Table 9.10. The raw data of query processing

time for the XMark dataset can be found in Appendix D. TwigPos was comparable to OPTwigPrime only when there is no much difference in the size of the intermediate storage built up by each algorithm. When *following-sibling* edges are present, OPTwigPrime significantly outperformed TwigPos as for PXQ_7. TwigPos outperformed OPTwigPrime in PXQ_3 and PXQ_5 due to the absence of pre-structural constraints and in PXQ_1 because there is only one match in the dataset. The reason is that OPTwigPrime uses a combination of the preorder and postorder filterings before storing elements in the intermediate storage which introduces some run-time overhead in order to skip irrelevant elements. Since the number of matching is quite small, the filtering has almost no effect, and only causes overhead.

Table 9.10: The overall comparisons based on U tests over the XMark dataset. "-" indicates no statistically difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
OPTwigPrime	5	3	0
TwigPos	3	5	0

To wrap up, OPTwigPrime showed a superior performance to the existing algorithm, TwigPos in terms of the space consumption in all queries and the query efficiency in some queries which are more complex than the rest of queries over this dataset. The performance ratio of OPTwigPrime over TwigPos ranges from 1.03 (with PXQ_7) to 36.13 (with PXQ_8). On the other hand, the performance ratio of TwigPos over OPTwigPrime ranges from 1.02 (with PXQ_5) to 6.61 (with PXQ_1).

9.4.2.2 TreeBank

Eight ordered and positional TPQs including complex and mixture of pre-structural and post-structural constraints (see Table 9.2) were used in the TreeBank dataset to show the differences between the algorithms where the XML document has a highly recursive structure. Figure 9.20 shows the size of the intermediate storage constructed by each algorithm. Clearly, OPTwigPrime stored by far fewer elements than TwigPos on over all queries tested on TreeBank. OPTwigPrime stored approximately one order of magnitude fewer elements than TwigPos due to the fact that it combines the efficient selection of useful elements for TPQs with a mix of P-C and A-D edges introduced in Chapter 6 and a strict subtree filtering match by one additional traversal over the intermediate results. This combination can handle pre-structural constraints efficently as shown in the experiments of Chapter 8. Interestingly, OPTwigPrime provided optimal evaluation for some queries such as PTQ_2 and PTQ_6.

To compare the query performance statistically, the experiment is based on hypothesis that there is no difference in the performance between the algorithms so that the Mann-Whitney U test was carried out to test that null hypothesis for each query. Thus, the total number of paired comparisons is equal to the number of queries because the experiment is

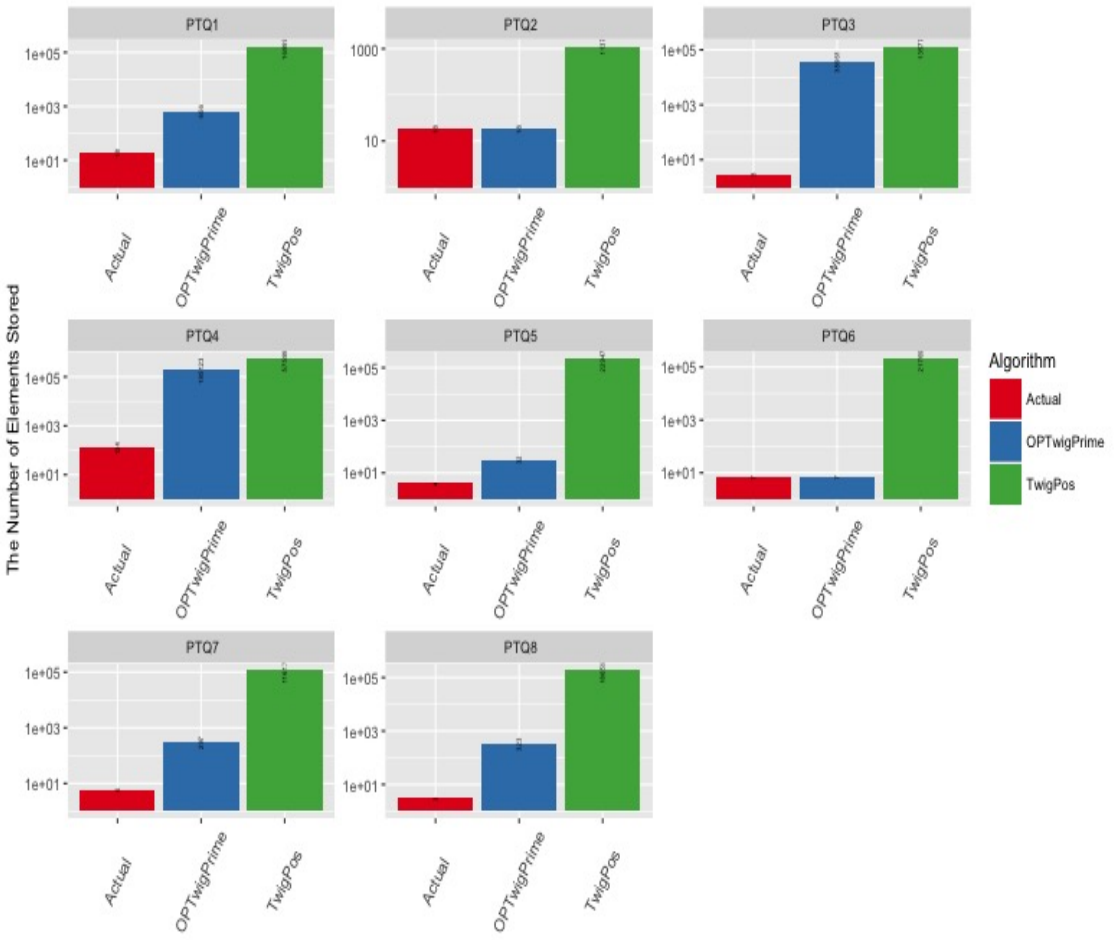


Figure 9.20: The number of elements stored in the intermediate storage by each algorithm for the ordered queries tested over the TreeBank document. "Actual" represents the number of elements relevant to the ordered query results.

limited to OPTwigPrime and TwigPos. All U tests turned out to reject the null hypothesis. Since there is a significant difference in the performance between the algorithms, the summary of the paired comparisons based on the Mann Whitney U test is provided in Table 9.11. The raw data of query processing time on TreeBank can be found in Appendix D. The performance result is plotted and fitted into Figure 9.21. OPTwigPrime significantly outperformed TwigPos in all queries except PTQ_4 because both stored a quite large number of irrelevant elements (see Figure 9.20). Another possible explanation for this is that only A-D edges are present during the preorder filtering match which are less restrictive while the P-C edge has to be checked by a strict postorder check.

Table 9.11: The overall comparisons based on U tests over the TreeBank dataset. "-" indicates no statistically difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
OPTwigPrime	7	1	0
TwigPos	1	7	0

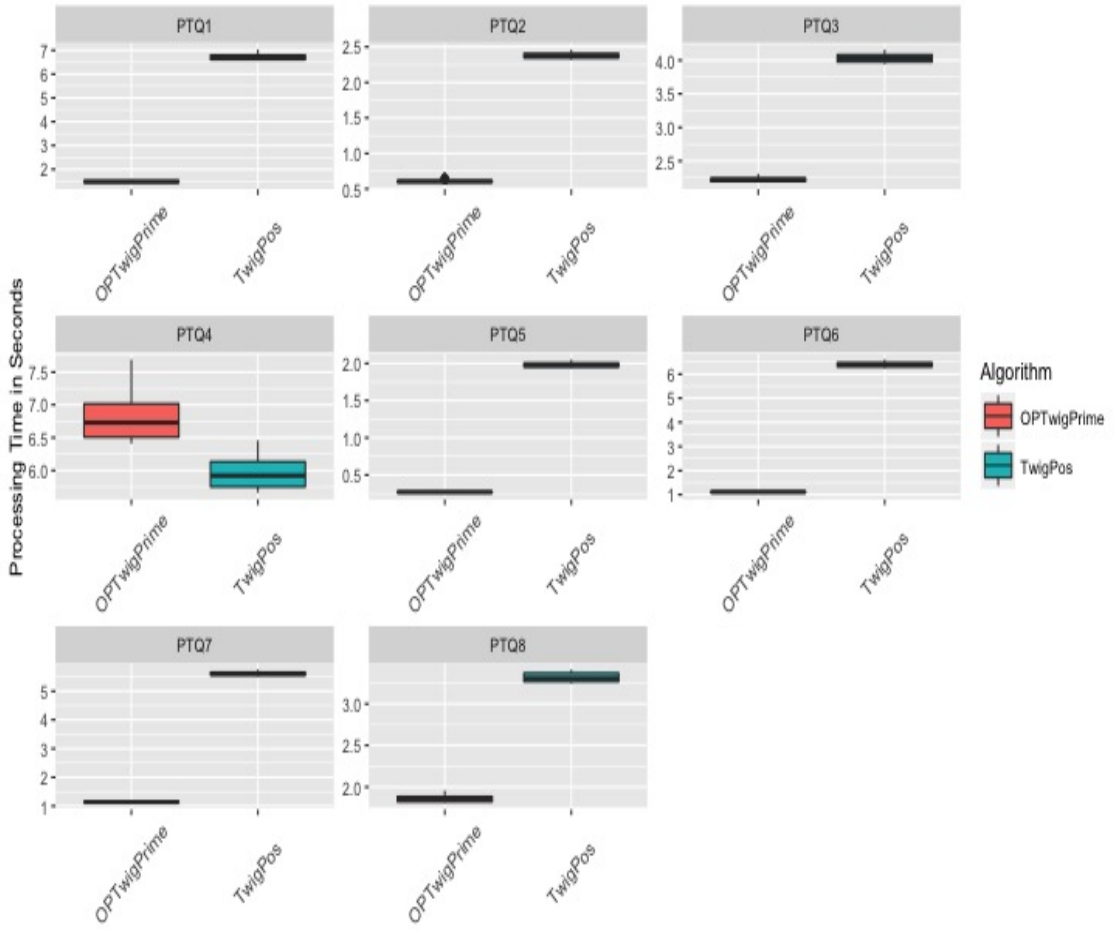


Figure 9.21: Query processing time of the algorithms compared for Ordered/Positional TPQs against TreeBank.

To sum up, OPTwigPrime significantly outperformed TwigPos in all queries (except PTQ_4, see Table 9.11). When the queries tested contain a combination of *following-sibling* edges and positional predicates, OPTwigPrime was up to five times faster than TwigPos. Overall, the improvement ratio of OPTwigPrime over TwigPos ranges from 1.80 (with PTQ_8) to 7.49 (with PTQ_5).

9.4.2.3 Random

The Random dataset has a complex structure with six distinct tags. This dataset was used to demonstrate the cost difference between OPTwigPrime and TwigPos because all elements are equally probable with a deeply recursive XML tree. The size of intermediate storage generated by each algorithm is presented in Figure 9.22. Generally, OPTwigPrime stored two orders of magnitude fewer elements than TwigPos over all queries on this dataset. For instance, in PRQ_7 and PRQ_8, the number of elements stored in OPTwigPrime was 105 and 61 times fewer than that stored in TwigPos, respectively. Note that PRQ_7 and PRQ_8 contain *following-sibling* axes which are supported by TwigPos.

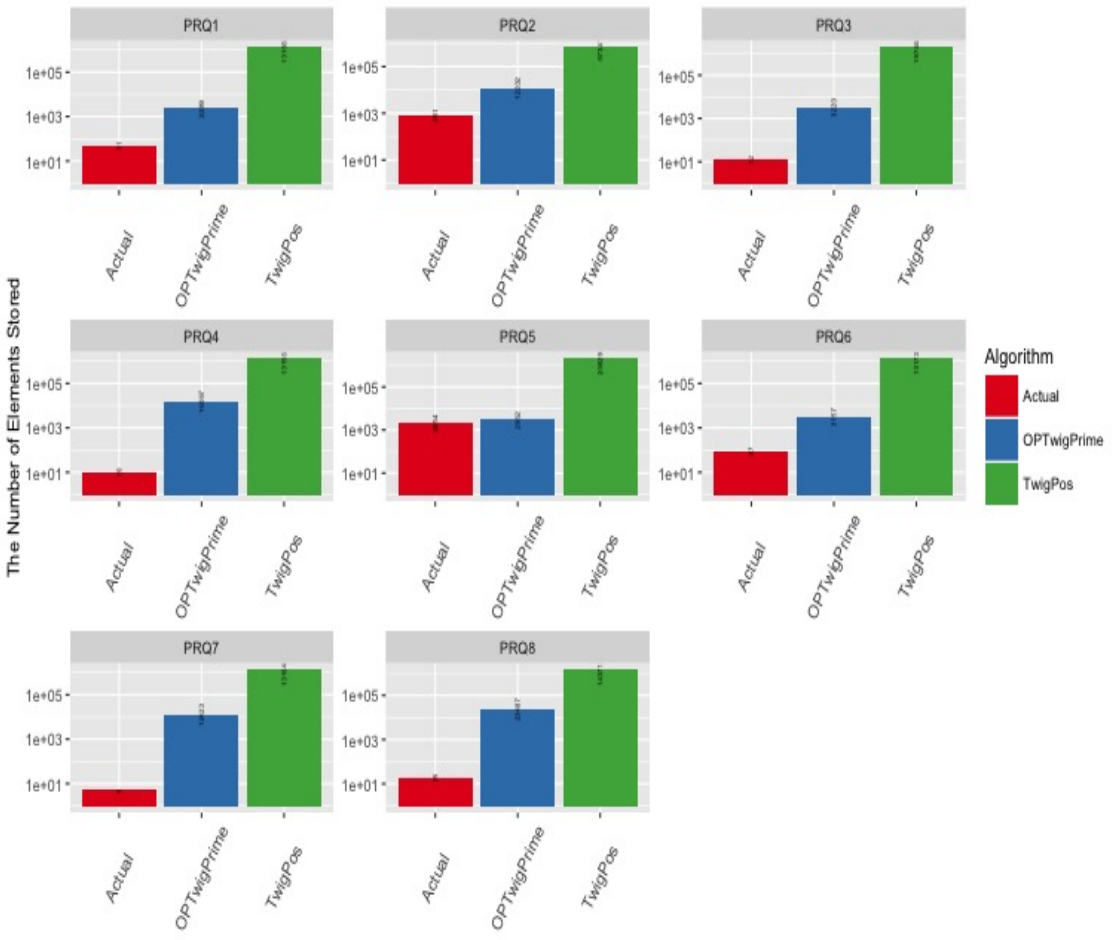


Figure 9.22: The number of elements stored in the intermediate storage by each algorithm for the ordered queries tested over the Random dataset. Actual represents the number of elements relevant to the ordered query results.

To compare the query efficiency, the Mann Whitney U test was carried out to test the null hypothesis for each TPQ tested. Since the results turned out to reject the null hypothesis by suggesting that for each TPQ there was a significantly difference in the performance between OPTwigPrime and TwigPos. The summary of the paired comparisons based on the Mann Whitney U test is given in Table 9.12. The full results are given in Appendix D. As reflected in the table, OPTwigPrime significantly outperformed TwigPos over all queries on the random dataset (see Figure 9.23).

In summary, OPTwigPrime showed a superior performance to TwigPos in terms of the number of elements stored in the intermediate storage and the CPU execution time.

Table 9.12: The overall comparisons based on U tests over the Random dataset. "-" indicates no statistically difference in the performance.

Algorithm	# of comparisons		
	Faster	Slower	-
OPTwigPrime	8	0	0
TwigPos	0	8	0

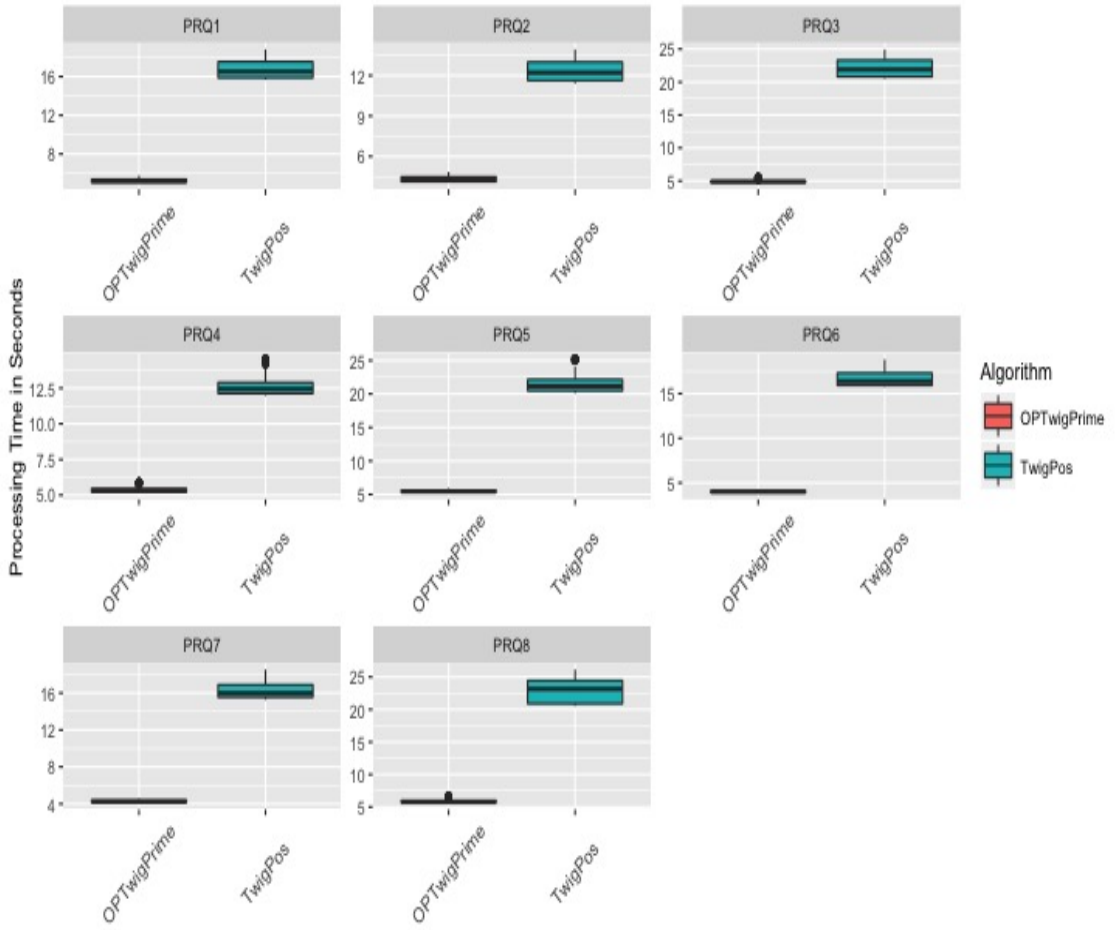


Figure 9.23: Query processing time of the algorithms compared for Ordered/Positional TPQs against the Random dataset.

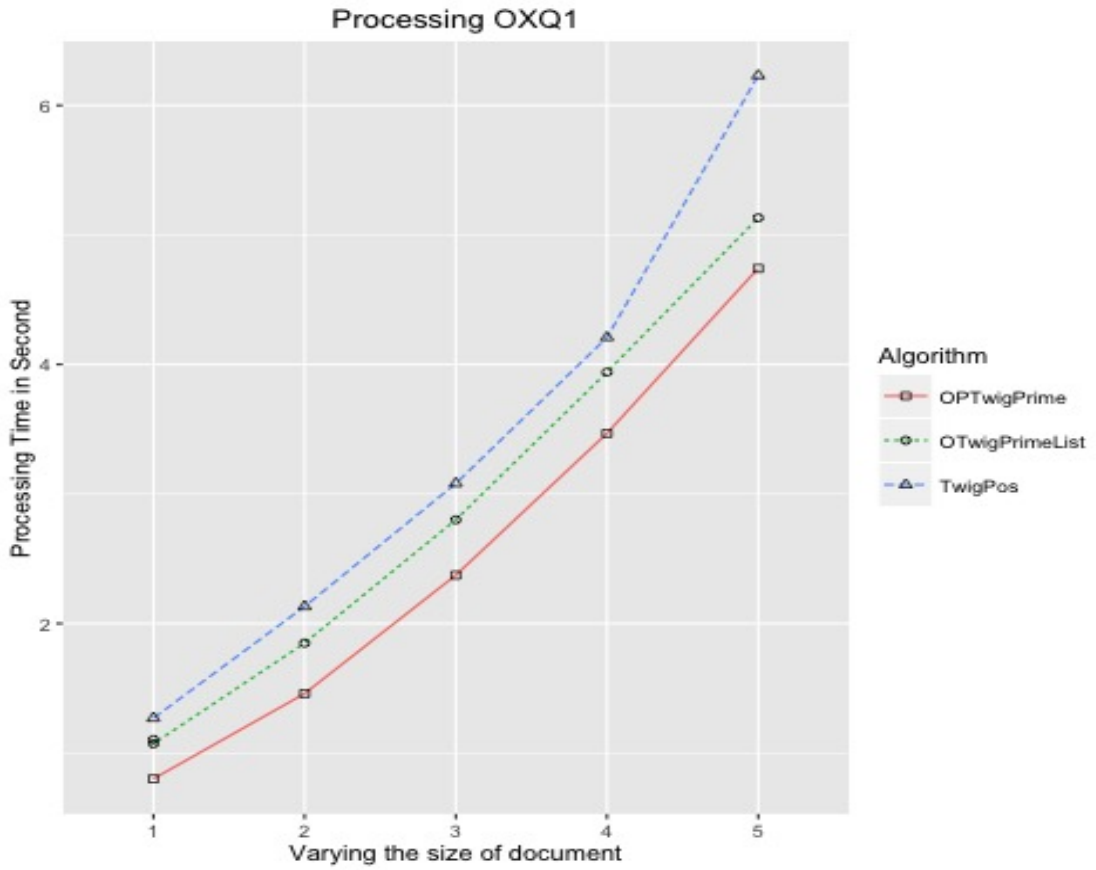
Overall, the improvement ratio of OPTwigPrime over TwigPos ranges from 2.42 (with PRQ_4) to 4.58 (with PRQ_3).

9.4.3 Scalability

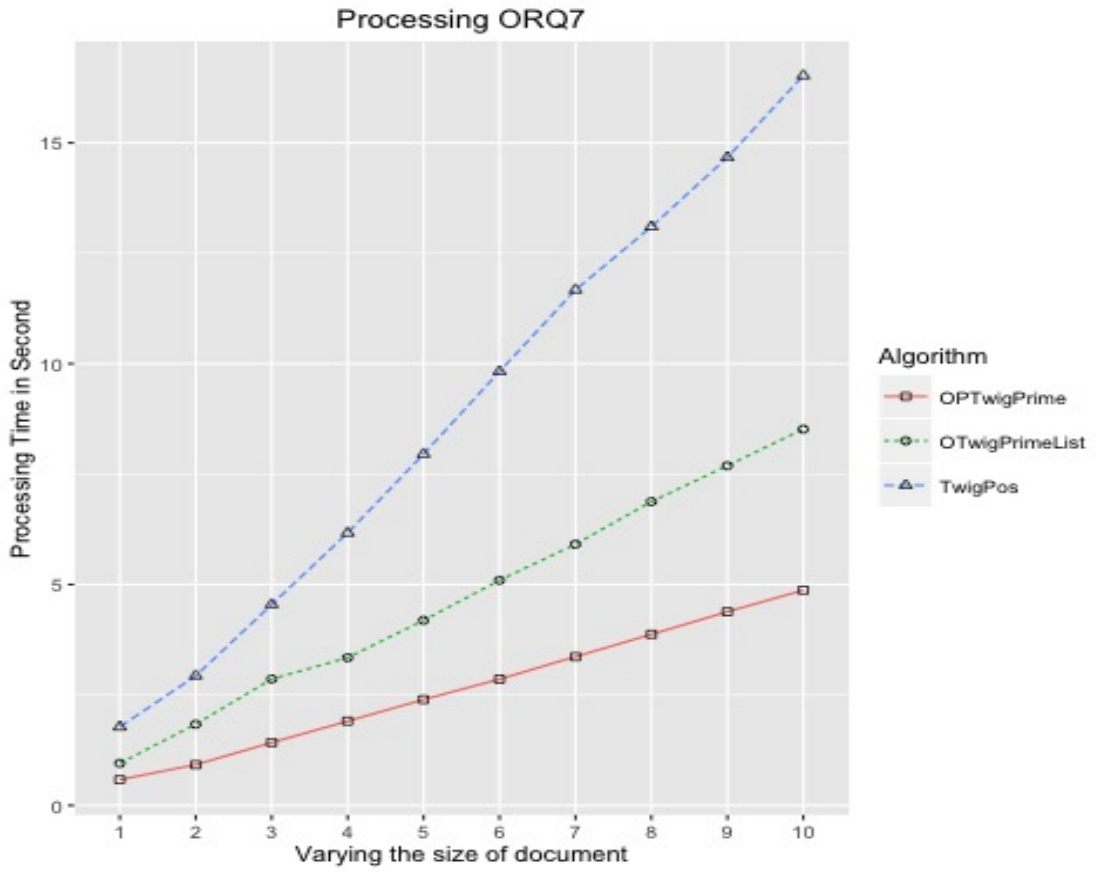
This section aims to evaluated the scalability of the new algorithms by increasing the size of the dataset. In this experiment, two datasets were used, the XMark and Random datasets. Whereas the XMark dataset is shallow and data oriented, the random collection has a deep recursive structure. Five different versions of XMark were created using the scaling factor from 1 to 5 as explained in Section 5.4.4. The Random dataset was partitioned into 10 different datasets to evaluate the scalability of the algorithms over irregular datasets (see Section 5.4.4). In order to make the experiment more objective, two TPQs have been selected over the two groups of datasets for each experiment in the query performance study (i.e., experiment 1 and 2), one of them being supported efficiently by TwigPos. As a result, OXQ_1 and PXQ_3 were selected for the XMark datasets, and ORQ_7 and PRQ_4 were chosen to be issued over the Random datasets.

The results for OXQ_1 and ORQ_7 are illustrated in Figure 9.24. With ordered TPQs, it can be observed that the new approaches scaled linearly with the increasing size of the dataset for both OXQ_1 and ORQ_7. On the other hand, for OXQ_1, TwigPos started to increase dramatically with the large datasets, and it has the worst scalability for ORQ_7.

Moving now to test the scalability with positional TPQs, the results are shown in Figure 9.25. The algorithms showed the same performance, they scaled almost linearly with the increasing size of the XMark document. However, when varying the size of the Random dataset. The reason for this is that OPTwigPrime stored 81 times fewer elements than TwigPos for PRQ_4.

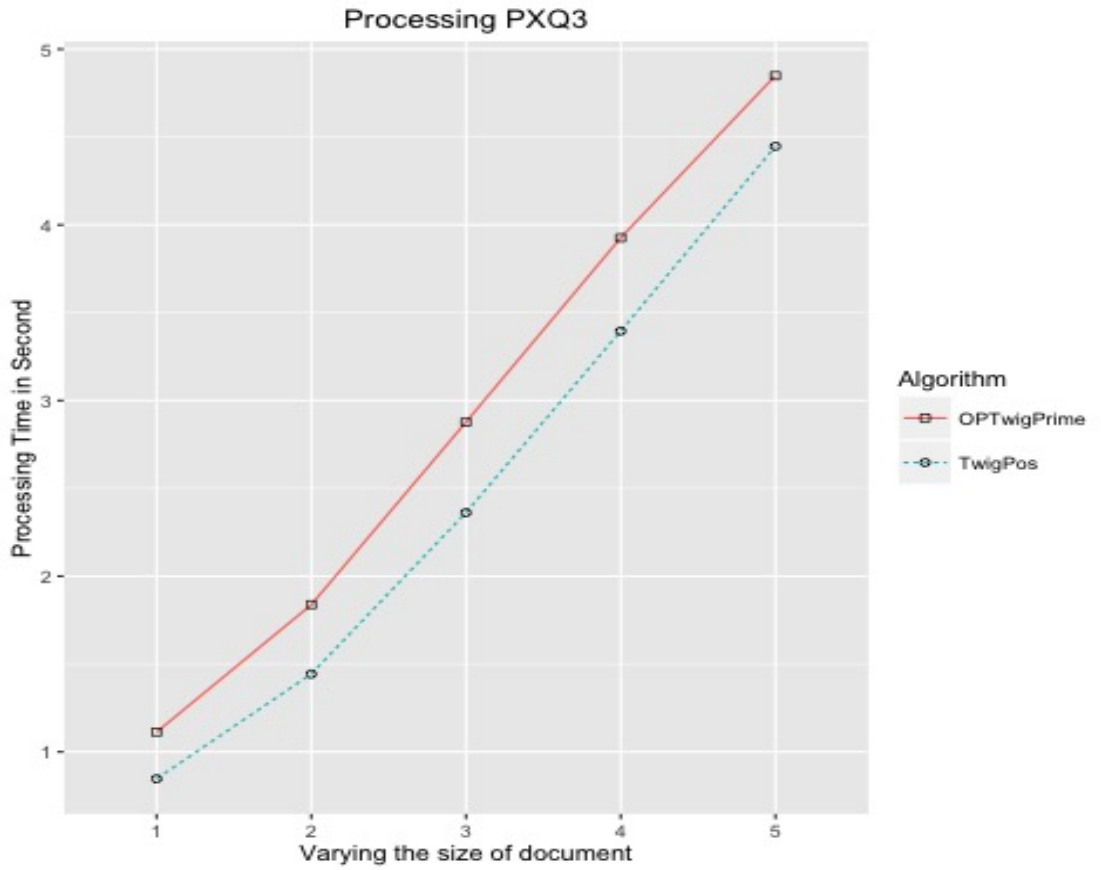


(a)

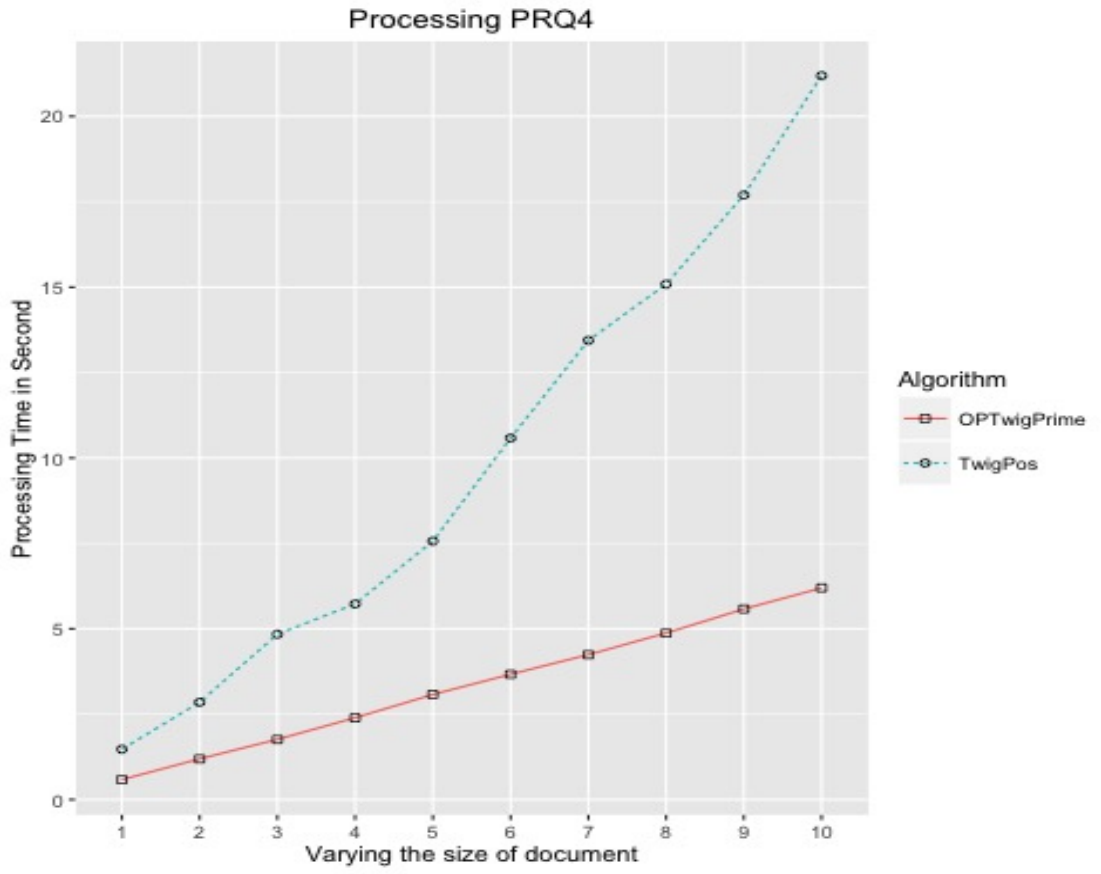


(b)

Figure 9.24: Scalability comparison for OXQ_1 and ORQ_7 against XMark and Random datasets, respectively.



(a)



(b)

Figure 9.25: Scalability comparison for PXQ_3 and PRQ_4 against XMark and Random datasets, respectively.

9.4.4 Summary

The aim of the present research was to explore the benefits of the early filtering for processing ordered and positional TPQs in bottom-up holistic twig matching algorithms. The experimental results have shown that the new approach can filter out many irrelevant elements effectively and it can be observed that the number of elements stored by the algorithms is significantly fewer than that stored by the existing approach. In addition, OPTwigPrime has the best performance in comparison to the rest of algorithms tested in most cases. The reason for this is that the new approaches' filtering lowers the cost of building intermediate results because their size is reduced and the cost of enumerating results because unnecessary traversal is avoided. The performance degradation of OTwigPrimeList when compared with OPTwigPrime is likely to be due to the expensive cost of fetching every element with ordering constraints into the buffering lists and performing additional filtering passes over the large buffering lists. Moreover, the scalability tests indicated that the new approaches have good scalability and they can scale well for large XML documents. An important observation is that TwigPos may output matching tuples unordered because it stores elements in postorder so that post-processing sorting of the output matches may be required to return query answers in preorder. On the other hand, the new approaches return the query results in document order. Finally, the experiments demonstrated the validity and improvement of the new approaches over the existing algorithm. These experiments also confirmed that the use of different filtering strategies does not cause any overhead in comparison to the existing approach. Thus, OPTwigPrime is the more powerful and most efficient bottom-up twig matching algorithm for processing ordered and positional TPQs.

9.5 Conclusion

In this chapter, two approaches, OTwigPrimeList and OPTwigPrime, to process ordered TPQs efficiently were proposed, without sequentially scanning the whole intermediate storage as in the case of top-down approaches. The first approach uses the advanced preorder filtering function introduced in OTJPrime to satisfy the SeqLR ordering constraints among TPQs with ordered axes or sequence operators by inspecting only head elements. The second approach makes use of the advanced preorder filtering function in OTJPrimeList. That is, some elements with order constraints are buffered in the main memory to avoid storing irrelevant elements so that a strict check for the three ordering constraints can be achieved. Unlike the first approach, the buffering algorithm can provide optimal enumeration in some cases as was shown in the experimental results. This is achieved by performing a strict check for the ordering constraints followed by a strict check for both prefix path and subtree matches. In addition, one approach has been devised to process TPQs with positional predicates. The new approach, OPTwigPrime combines features of previous approaches to process TPQs such as the CPL approach, level split

intermediate storage and the state-of-the-art advanced preorder filtering function, *getMatch*. OPTwigPrime is the first of its kind using preorder and postorder filtering for processing ordered and positional TPQs.

The following chapter describes the overall evaluations of the experimental results obtained in this thesis in the light of the research hypothesis, and identifies the limitations of the experiments' design. Furthermore, the main findings of the CPL approach for processing structural XML queries will be given.

Chapter 10

The Overall Evaluation

10.1 Introduction

In Chapters 6, 7, 8 and 9, the design of four experiments was described; each experiment evaluated a set of XML query processing algorithms in terms of query coverage, performance and scalability. The first experiment was designed to test top-down holistic twig matching algorithms which process and store XML elements in preorder and use the output enumeration introduced in TwigStack. This experiment focused on a subset of XPath expressions, which contains Parent-Child (P-C) and Ancestor-Descendant (A-D) relationships since this type of query is the main building block for more complex queries. The second experiment evaluated the new top-down holistic approaches for processing ordered queries which contain ordering constraints in addition to the basic axes. Furthermore, the third experiment was limited to the bottom-up category of holistic twig matching algorithms on the subclass of TPQs which contains only P-C and A-D edges and use the output enumeration introduced in TwigList. Finally, the last experiment tested bottom-up holistic twig algorithms which are capable of processing TPQs with ordering constraints and positional predicates.

The results of the four experiments were provided and analysed in Chapters 6, 7, 8 and 9, respectively. However, the results will be reconsidered here from different perspective in order to provide a more comprehensive evaluation of the new approaches. The specific objective of this chapter is to evaluate the experimental results in the light of the research hypothesis stated in Chapter 4. With this intention, each experiment's results are discussed in an individual section.

The rest of this chapter is organised as follows. Section 10.2 highlights the main aim of the experiments. To achieve this goal, several procedures were followed while designing the experiments and they are presented in Section 10.2.1. Moreover, Section 10.3 presents the overall evaluation of the results for each experiment individually. After that, features and limitations of the experiments are summarised in Section 10.4. As a final point, the experiments' main findings will be discussed in Section 10.5. The chapter is concluded in Section 10.6

10.2 The Objective of the Experiments

One purpose of the experiments conducted in Chapters 6, 7, 8 and 9, was to establish a fair comparison between the relative performance of the new algorithms proposed in this thesis and the up-to-date approaches in the literature. Many existing algorithms have been proposed in the literature, of which only relevant approaches have been selected to provide a valid comparison. The algorithms compared were chosen to represent different groups of holistic twig matching algorithms. The main aim of the four experiments is to demonstrate the validity of the new approaches. As a result, throughout this thesis, the experimental evaluations were carried out to test the research hypothesis which was stated in Chapter 4.

> **"Encoding names of child elements of branching XML's elements based on Parent-Child relationships may improve the efficiency of holistic twig join algorithms by increasing the query processor's coverage as well as reducing computation cost and memory consumption."**

Before proceeding to the next section, the criteria to compare the chosen algorithms with the new approaches are described and they are based on the following aspects: the data structure utilized to store potential elements (i.e., top-down approaches in contrast to bottom-up approaches), the underlying indexing technique (i.e., range-based labelling scheme), the I/O mechanism where the data must be scanned only once in a sequential read (i.e, holistic approaches) and the filtering strategy (i.e., top-down in contrast to bottom-up). Furthermore, the matching output must be returned as a set of tuples representing all matching elements corresponding to query nodes according to Definition 4.11. The next section restates the guidelines which were taken into account while running the experiments to ensure the completeness and correctness of the experimental evaluation [159, 200].

10.2.1 The Strategy of the Experimental Evaluation

In order to achieve the objective of the experiments, some constraints, which emerge from the nature of computer science as a combination of science and engineering, to be followed during the design of the experiments in this thesis [159, 202, 200, 37, 156]. The following steps were conducted to fulfil the main objectives and assure the reliability of the conclusions:

- The appropriate implementations where the new approaches were implemented and evaluated under the available Hardware/Software, in an environment where the implementations of the algorithms to be compared is as similar as possible must be identified. Therefore, a fair comparison can be guaranteed. This was covered in Chapter 5.

- The criteria for selecting the datasets used in the experiments and the structured XML queries issued over them must be clear. In addition, the selected datasets and XML query sets must show their strengths and weaknesses. The ability to investigate the

limitations corresponding to each set of XML queries and datasets must be available as well. For the purpose of scalability tests, at least two datasets with significantly different schemas must be used to avoid the potential bias of using a single dataset or datasets with similar structure. This was given in Chapters 5, 6, 7, and 9.

- For each experiment, the metrics must be identified and unified throughout the entire evaluation for all the approaches compared. More details can be found in Chapters 6, 7, 8, and 9.

- The platform setup in which the experiments take place must be described including the Hardware and Software specifications. This was provided in Chapter 5.

- The use of statistical tests must be described in order to analyse properly the results obtained from the experiments. The research thus provides conclusions with statistical significances. This was provided in Chapter 5.

This section summarised the strategy of the experimental evaluation in this thesis which was described in Chapters 5, 6, 7, 8 and 9.

10.3 Evaluation From Different Perspective

This section describes the evaluation of a novel set of holistic twig join algorithms based on the **Child Prime Label** (CPL) approach for processing TPQs in native XML databases (see Chapter 5). These approaches aimed to improve the efficiency of holistic twig matching algorithms and extending holistic querying algorithms to make them able to handle ordering constraints and positional predicates in XPath expressions. The research study achieved the main objectives for the new solution by incorporating encoded names of children into the existing labelling schemes with minimal computation and space overheads. Based on this novel mechanism to label XML elements, the thesis proposed a set of novel holistic twig matching algorithms which are easily capable of processing different subclasses of TPQs including P-C relationships, A-D edges, ordered axes and positional predicates.

To test the research hypothesis, the *holistic* model was successfully implemented. The design of the full implementation was described in Chapter 5. The node labelling scheme was extended to use the CPL approach. Furthermore, the query processor was augmented with the up-to-date algorithms and a set of novel algorithms which were introduced in Chapters 6, 7, 8, and 9. For each experiment, the experimental results were analysed after the description of the experimental design. The evaluation of the performance was subsequently provided. Also, the experiments examined the behaviour of the algorithms compared over different groups of datasets in order to test their scalability.

In the four experiments, two variables were measured during the run of each TPQ over the XML datasets: the processing time taken by each algorithm to output the query matches in milliseconds and the number of paths or elements stored by each algorithm when answering the query. To increase the reliability of the measures, all TPQs were

warmed up by three runs and then each TPQ was executed 100 times (see Section 5.5). It is interesting to note that in all four experiments of this study, the research hypothesis was supported by empirical evidence based on the proposals, the experimental design and results. Generally, the CPL approaches (i.e., holistic twig matching algorithms based on the CPL approach) showed a superior performance to the existing algorithms in terms of query running time and the intermediate storage size.

The aim of this section is to evaluate the experimental design and results in order to identify further improvements. To facilitate the evaluation, the improved ratio of algorithm pairs for all TPQs over each dataset in the experiments will be used to demonstrate the scale of the improvement of the new approaches over the algorithms compared. The improved ratio (IR) of algorithm A over algorithm B for a set of queries issued over an XML dataset can be computed using Formula 10.1 [130, 22], where T_A and T_B are the median running times for algorithms A and B, respectively.

$$IR_{A,B} = \frac{T_B - T_A}{T_B} \tag{10.1}$$

The main attribute indicating the efficiency of every holistic twig matching algorithm related to I/O complexity is the intermediate storage size. For the first and second experiments reported in Chapters 6 and 7, the intermediate result size for each algorithm will be evaluated by computing a ratio of the number of paths stored by each algorithm and the number of relevant paths for each dataset, whereas for the third and fourth experiments discussed in Chapters 8 and 9, the intermediate result size for each algorithm will be evaluated by computing a ratio of the number of elements stored by each algorithm and the number of relevant elements for each collection. If the ratio has value that equals to 1, then the algorithm was optimal for all queries tested for that dataset. Otherwise, the smaller the value of the ratio the better, since this indicates that the algorithm was more successful in filtering out irrelevant elements to queries tested on a specific collection.

The subsequent sections evaluate all the experiments conducted in this thesis in terms of their designs and results using these new measures

10.3.1 Top-Down Approaches for the Basic Structural Axes

This section evaluates the design and results of the first set of experiments which focused on incorporating the notion of CPL approach into the existing labelling scheme (range-based labelling scheme was chosen because it is used by most holistic twig matching algorithms related to this thesis since it has commonly been assumed that integer comparisons are several times faster than string comparisons [102]) and determining the effect of the CPL approach to improve the filtering strategy of twig matching algorithms by comparing the performance of TwigStackPrime against the performance of two representative top-down algorithms from the literature. The results, as expected, showed that the CPL approach can be successfully added to the existing labelling scheme without affecting the functionality

of the labelling scheme and consuming a large amount of storage in disk-resident data structure (i.e., streams of elements). Although five datasets with different structures, sizes and number of tags were used to test the augmentation of the CPL approach, the maximum increase in the size of the indexing files was only 5 MB (see Section 6.5). The optimised approach to assign CPL attributes showed no sign of overflow (e.g., the maximum number of distinct tags used in the experiment was 251). Figure 10.1 shows the summary of the results for this experiment. Overall, these results indicate that the CPL approach (i.e., TwigStackPrime) provided an efficient solution to reduce storage space by skipping irrelevant elements from the input streams (see Figure 10.1c) and to improve the overall performance (see Figure 10.1b). By way of illustration, on the Random dataset, the improvement of TwigStackPrime over TwigStackList, denoted as $IR_{TwigStackPrime,TwigStackList}$, is more than 25%. Furthermore, TwigStackPrime showed basically linear scalability.

DBLP	XMark	TreeBank	Random
TwigStackPrime	TwigStackPrime	TwigStackPrime	TwigStackPrime
TwigStack	TwigStackList	TwigStackList	TwigStackList
TwigStackList	TwigStack	TwigStack	TwigStack

(a) Compared algorithms sorted according to their IRs

	DBLP	XMark	TreeBank	Random
TwigStackList	0.122	0.005	0.220	0.256
TwigStack	0.081	0.076	0.644	0.740

(b) The IR of TwigStackPrime compared to all approaches tested for all queries for each collection

	DBLP	XMark	TreeBank	Random
TwigStack	1.037257	1.25277	10.73372	2.810589
TwigStackList	1	1	1.122552	1.252916
TwigStackPrime	1	1	1.001484	1.007868

(c) Ratio of the number of paths stored in intermediate storage and the number of relevant paths for each dataset

Figure 10.1: The experimental results of top-down approaches for TPQs with {/,//,[]}.

In this experiment, all path solutions were stored in main memory. However, when the intermediate storage sets are large and can not fit into main memory, they must be written to and read in from secondary storage. This means that TwigStackPrime is expected to further outperform TwigStack and TwigStackList when I/O operations are counted into the total processing time. Since the CPL approach is not dependent on a specific type of labelling scheme, it can be used with any labelling scheme. Thus, the experiment could be extended to study the impact of the CPL approach when incorporated into prefix-based path labelling schemes such as extended Dewey [147]. The process of assigning CPL attributes could be further optimised by using statistics relating to inner elements in the XML data. Notice

that this is different from the simple information obtained when counting the number of sibling-unique tag names introduced in this thesis (see Chapter 6).

10.3.2 Top-Down Approaches for Ordered Constraints and Positional Predicates

The experiment was designed to test the performance of the new approaches which are easily capable of processing ordered axes[1] and sequence operators by representing effectively XPath expressions with ordered axes and sequence operators using tree structures. Because the new approaches, unlike all other holistic approaches, are capable of working with order-aware TPQs, the experiment compared the performance of the new approaches against the existing top-down algorithms which were extended to process ordering constraints during the output enumeration (i.e., the second phase of top-down approaches). Figure 10.2 shows the summary statistics for this experiment. Together these results provide important insights into the effect of the CPL approach in enlarging the query processor's coverage in holistic approaches. The buffering algorithms showed a superior performance to the rest of algorithms tested because they use combined approaches and perform semi-strict filtering checks in both vertical and horizontal relationships. However, their performance could be further improved by avoiding linear search over the large buffering lists. This can be achieved if every element is compared to the first and last element in the corresponding preceding/following lists since they are sufficient for a semi-strict filtering match. One of the greatest challenges when evaluating order axes and sequence operators is that elements which have satisfied P-C and A-D relationships might not satisfy order axes. However, the new buffering approaches (i.e., OTJPrimeList and OTJPrimeMultiLists) provided nearly optimal evaluation for all collections with respect to the intermediate result size. Moreover, considering only the minimal constraint check among head elements resulted in significantly lowering the number of useless paths in the case of OTJPrime. The filtering could be improved if elements are further checked in the main algorithm to avoid storing non-contributing elements. The scalability tests indicated that the querying time needed for the new approaches is linearly correlated to the dataset size. That is, with the increase of dataset size, the benefit of the new approaches over the existing algorithms correspondingly increases.

Similar to the previous experiment (see Section 10.3.1), all root-to-leaf path solutions were kept in the main memory. However, the cost difference between the algorithms compared will be huge if the path solutions are larger than the available main memory. The results obtained from this experiment form a base for further extensive tests, nevertheless more reliable results could be obtained by extending the query set and/or incorporating the ideas of ordered and descendant extension into the existing top-down algorithms.

[1] order axes = $\{preceding,\ following,\ preceding-sibling\ and\ following-sibling\}$

XMark	TreeBank	Random
OTJPrimeList	OTJPrimeList	OTJPrimeList
OTJPMultiLists	OTJPMultiLists	OTJPMultiLists
OTJPrime	OTJPrime	OTJPrime
SFTwigStackList	SFTwigStackPrime	SFTwigStackPrime
SFTwigStackPrime	SFTwigStackList	SFTwigStackList
SFTwigStack	SFTwigStack	SFTwigStack

(a) Tested algorithms sorted according to their IRs

	XMark	TreeBank	Random
SFTwigStack	0.942	0.662	0.933
SFTwigStackList	0.941	0.648	0.817
SFTwigStackPrime	0.941	0.539	0.814
OTJPrime	0.496	0.051	0.520
OTJPMultiLists	0.010	0.023	0.088

(b) The IR of OTJPrimeList compared to all approaches tested for all queries for each collection

	XMark	TreeBank	Random
OTJPrimeList	1	1.00083	1.169596
OTJPrimeMultiLists	1	1.00083	1.169596
OTJPrime	1.681548	1.558014	3.76566
SFTwigStackPrime	4.257155	2.553089	6.530527
SFTwigStackList	4.257155	2.811911	7.15333
SFTwigStack	4.312132	3.06518	13.66382

(c) Ratio of the number of paths stored in intermediate storage and the number of relevant paths for each dataset

Figure 10.2: The experimental results of top-down approaches for TPQs with ordered axes and sequence operators.

10.3.3 Bottom-Up Approaches for the Basic Structural Axes

Motivated by the success in efficient filtering of irrelevant elements using the CPL approach for $\text{TPQ}^{\{/,//,[]\}}$, the experiment evaluated the performance of a set of new approaches which are based on the CPL approach and use a combination of preorder and postorder filtering matches against the state-of-the-art holistic twig matching algorithms. The experimental design and results were respectable but they could include more bottom-up algorithms. Some algorithms were not considered in the experiment because it had been reported that the algorithms used in the experiment significantly outperformed them. For instance, [185] reported that TwigList was several times faster than $\text{Twig}^2\text{Stack}$; similarly [132, 89] reported that TwigFast outperformed HolisticTwigStack for queries similar to that used in this experiment. Figure 10.3 provides a summary obtained from the analysis of the experimental results. It can be seen from the data in Figure 10.3 that the CPL algorithms significantly outperformed the other up-to-date methods. A nearly optimal evaluation has been provided in complex datasets with many recursions in the structure and an

optimal evaluation has been provided for relatively structured XML collections with a lot of repetitive structures (see Figure 10.3c).

The query performance of TwigPrimeMatch can be further improved by maintaining, for each inner query node, the latest ancestor in tree postorder that has been returned by *getMatch* in previous calls similar to that used in *getPart*. This could significantly reduce the number of iterations required to check whether or not there is a relevant ancestor in the intermediate storage. Moreover, the scalability test was performed with varying the size of datasets and showed that the new approaches are outstanding in their linear scalability. However, the scalability test could be extended to not only vary the sizes of dataset but also vary the number of P-C axes in the queries to further investigate the behaviour of the CPL approach.

Due to time constraints the GTP semantics (see Chapter 3) were ignored (i.e., all query nodes are considered as output nodes and TPQs do not contain OR and NOT operators) but no difficulties in including them are anticipated because they are less restrictive predicates than AND operators which have been successfully implemented. This issue can be considered as future work to provide more comparative results when comparing GTPStack with the proposals for processing GTPs. An important observation is that the bottom-up approaches overcome the main weakness of top-down methods which can be attributed to their inefficient output enumeration (i.e., the enumeration algorithm reads the entire output arrays even though useless paths are stored in the intermediate storage). Conversely, the bottom-up approaches suffered from a high memory usage. The experiment could be extended to test extensively the benefit of early enumeration approach to alleviate the main memory consumption when the intermediate storage kept in the main memory. On the other hand, I/O costs should be included when the intermediate storage exceeds the memory limit of a stand-alone computer.

TJStrictPostPrime_		TwigPrimeMatch_N			
	DBLP		XMark	TreeBank	Random
TwigList	0.845392	TwigList	0.273789	0.886956	0.804136
TwigFast	0.595187	TwigFast	0.338623	0.887777	0.733736
TwigPrime_	0.647691	TwigPrime_	0.550353	0.814618	0.576148
TwigPrime_N	0.647880	TwigPrime_N	0.533485	0.810421	0.571815
TwigPrime	0.662291	TwigPrime	0.576910	0.820713	0.604048
TwigPrimePart_	0.001522	TwigPrimePart_	0.497283	0.121738	0.176654
TwigPrimePart_N	0.001522	TwigPrimePart_N	0.496382	0.113134	0.176479
TwigPrimeMatch_	0.047896	TwigPrimeMatch_	0.000264	0.009731	0.012845
TwigPrimeMatch_N	0.031019	TwigPrimePart	0.418207	0.094694	0.102553
TwigPrimePart	0.004552	TwigPrimeMatch	0.479305	0.779424	0.563903
TwigPrimeMatch	0.631978	TJStrictPre	0.112333	0.731077	0.138516
TJStrictPre	0.197062	TJStrictPre_	0.031904	0.725904	0.142467
TJStrictPre_	0.201461	TJStrictPrePrime	0.128947	0.058569	0.085152
TJStrictPrePrime	0.004552	TJStrictPrePrime_	0.075104	0.046249	0.082390
TJStrictPrePrime_	0.001522	TJStrictPost	0.053643	0.731177	0.143128
TJStrictPost	0.200974	TJStrictPost_	0.000395	0.733055	0.144189
TJStrictPost_	0.199512	TJStrictPostPrime	0.084480	0.065729	0.078033
TJStrictPostPrime	0.004552	TJStrictPostPrime_	0.038530	0.062163	0.070078
GTPStack	0.189122	GTPStack	0.721175	0.982632	0.916856
GTPStackPrime	0.053391	GTPStackPrime	0.806992	0.957722	0.821266

(a) The IR of TJStrictPostPrime_ compared to all approaches tested for all queries on DBLP and the IR of TwigPrimeMatch_N compared to all algorithms tested for all queries on XMark, TreeBank and Random

TJStrictPrePrime	
	Zipf
TwigPrime	0.536
TwigPrimePart	0.044
TwigPrimeMatch	0.484
TJStrictPre	0.528
GTPStack	0.986
GTPStackPrime	0.921

(b) The IR of TJStrictPrePrime compared to all approaches tested for all queries on Zipf

	DBLP	XMark	TreeBank	Random	Zipf
TwigFast	1.590525	1.195006	16.90548	4.815722	
TwigList	577.9526	6.347786	34.42001	65.00936	
CPL_	1	1	1.036419	1.009365	
CPL	1	1	1.03293	1.009345	1.000043
TJStrictPost	1.275804	1.021521	6.051748	2.255521	
TJStrictPost_	1.275804	1.021521	6.180917	2.279508	
TJStrictPre	1.275804	1.021521	6.051799	2.255621	4.129953
TJStrictPre_	1.409475	1.087964	9.929092	2.650552	
GTPStack	1.275804	1.021521	6.051799	2.255621	4.129953

(c) Ratio of the number of elements stored in intermediate storage and the number of relevant elements for each dataset

Figure 10.3: The experimental results of bottom-up approaches for TPQs which use only the P-C and A-D axis.

10.3.4 Bottom-Up Approaches for Ordered Constraints and Positional Predicates

The last set of experiments was designed to compare the performance of the new bottom-up approaches for processing a subset of twigs which may contain ordered axes, sequence operators and positional predicates. The new bottom-up methods are based on the preorder filtering function for P-C edges and ordering constraints introduced in Chapter 7. The first experiment focused on algorithms which can provide efficient evaluation of ordering constraints. The results obtained from this experiment are shown in Figure 10.4. Surprisingly, the simple filtering check of ordering constraints performed by OPTwigPrime significantly outperformed the other methods tested. The cost difference between OPTwigPrime and OTwigPrimeList is due to the fact that the latter has to perform some extra operations to avoid storing useless elements. In connection with the issues mentioned in Section 10.3.2, the benefit of filtering out irrelevant elements in the bottom-up approach exceeded by the in-memory cost of caching and handling (i.e., linear search) elements with ordering constraints in the buffering lists.

XMark	TreeBank	Random	
OPTwigPrime	OPTwigPrime	OPTwigPrime	
OTwigPrimeList	OTwigPrimeList	OTwigPrimeList	
TwigPos	TwigPos	TwigPos	
	XMark	TreeBank	Random
OTwigPrimeList	0.271	0.501	0.595
TwigPos	0.412	0.647	0.889

(a) The top table presents algorithms sorted according to their IRsand the bottom table shows the IR of OPTwigPrime compared to all algorithms tested for all queries for each collection

	XMark	TreeBank	Random
OTwigPrimeList	1	1.007719	1.248917
OPTwigPrime	2.188501	1.670205	3.712064
TwigPos	11.43852	18.32452	104.598

(b) Ratio of the number of elements stored in intermediate storage and the number of relevant elements for each dataset

Figure 10.4: The experimental results of bottom-up approaches for OTPQs.

The second experiment evaluated the comparisons between OPTwigPrime and TwigPos because there is no other holistic approach which is capable of processing positional predicates. Figure 10.5 compares the summary statistics for this experiment. Even though OPTwigPrime significantly outperformed TwigPos for all collections, its performance could be further improved by considering the subclass of TPQs which falls within the optimal group of the CPL approach during the filtering phase. To put it another way, if the sub-twig query built from query nodes specified in the pre-structural constraints contains

only P-C edges related to leaf query nodes, then irrelevant elements can be discarded and *mismatch* tables can be updated safely.

	XMark	TreeBank	Random
TwigPos	0.585	0.575	0.720

(a) The IR of OPTwigPrime compared to TwigPos for all queries

	XMark	TreeBank	Random
OPTwigPrime	2.414471	1231.787	24.5489
TwigPos	5.491509	8922.656	3702.741

(b) Ratio of the number of elements stored in intermediate storage and the number of relevant elements for each dataset

Figure 10.5: The experimental results of bottom-up approaches for ordered and positional TPQs.

The scalability test was done with increasing dataset sizes. It revealed that the new approaches demonstrated linear scalability of the running time with the queries tested. The experiments would have been better if they had included more algorithms, datasets and queries. However, the performance comparison was limited to TwigPos because it is the only, to date, bottom-up algorithm which can support some of ordered axes and positional predicates in the filtering phase. The experiment could be extended to study the effect of the ordered child and descendant concept when incorporated into the existing holistic methods.

10.4 Features and Limitations of the Experiments

The experiments conducted in this thesis were designed to measure specifically whether or not the hypothesis stated in Chapter 4 can be accepted. They were intended to be as simple as possible and effective. However, the comparison experiment is susceptible to two problems: a bias in favouring of the proposals and a selection for incomparable approaches, thus standard precautions were strictly followed, during the design of these experiments, to improve the validity of the comparison. Due to the time constraints, the outcomes of these experiments are limited in a number of ways discussed in the following subsections. While the features of the experiments are described in Section 10.4.1, the limitations of the experiments are highlighted in Section 10.4.2.

10.4.1 Features of the Experiments

- The experiments were conducted over several datasets each of which has a different schema. The main goal of this is to test the influence of different XML structures on the algorithms compared. The selected datasets include XML data from various resources such as real-world, benchmark and synthetic. The use of real-world and

benchmark data was to demonstrate the performance of the algorithms in practice whereas the synthetic data allowed to cover some aspects that may not be found in the real-world data.

- Numerous queries were used in the experiments, most of them were used by the algorithms compared in experiments conducted in previous work, but the number was still limited.
- The experiments were thorough in the sense that they considered two major components of native XML databases (NXDs): storage engine and query processor.
- The experiments measured the main variables that have a significant relationship to the query performance in native XML database management systems (NXDMSs).
- The experimental design included the plan for analysing and reporting of the results. It was made available (see Chapter 5).
- More datasets, queries and algorithms could be included seamlessly in the framework to obtain more elaborate results.
- The full experimental results, which are easy to interpret, can be found in Appendices A, B, C and D for further and related research.

10.4.2 Limitations of the Experiments

- Since the current study was limited to the searching of TPQ matches in an XML tree, it was not possible to support the semantics of XPath. In XPath there is only a single query node which can be returned whilst twig pattern matching outputs all legal combinations of matches which is useful for the flexibility of XQuery. However, the query performance could be tested to return a specific query output node when evaluating less complicated XPath queries.
- It is unfortunate that the experiments did not consider the GTP semantics and content-based queries. Also, they did not address a subset of TPQs which involve wildcards "*" and logical operators.
- The experiments did not make use of indices (e.g., XB-tree) built over the input streams in order to further speed up the query running time by reducing I/O costs.
- The intermediate storage was stored in the main memory. However, in order to test whether the holistic algorithms would be able to exploit fully the available size of main memory when the intermediate storage size is larger than the available memory, the intermediate results must be written to and read from the secondary storage.

10.5 The Main Findings of the Experiments

The most obvious finding to emerge from the experimental results is that the main idea of research hypothesis was confirmed. Generally, the experiments showed that the CPL approach can be incorporated into the existing labelling schemes without any additional overhead. Through them, the validity and improvements of the new approaches over the other related algorithms can be demonstrated. The scalability analysis revealed that the new approaches can efficiently scale up, since when the size of dataset increases, non-contributing elements do not proportionally consume more processing time because the optimal optimization that the CPL approach alone possesses. These findings suggest that in general the new algorithms are more suitable for processing large XML documents than the existing methods. Moreover, this is the first study to investigate the effect of early filtering for ordering constraints and positional predicates in a single algorithmic framework. This thesis has raised important questions about the amount of information encoded within XML labels to provide an efficient evaluation of XML queries. The main findings (contributions) of the whole research study will be discussed in the next chapter (see Chapter 11).

10.6 Conclusion

This chapter presented an overall evaluation of the designs and results of the experiments carried out in this research study. The experimental results were reconsidered to allow a precise comparison for each XML collection. During the evaluation, some suggestions and improvements were briefly discussed. They will be further explained as recommendations for further research work in the next chapter. In addition, a description of the features, limitations and findings of the experiments was given.

All in all, the findings of this thesis could be used to draw the following conclusion. The CPL approach is a new source of improvement for holistic twig matching algorithms since it can reduce the number of elements processed and the size of intermediate result when TPQs contain Parent-Child edges. Through experiments the validity and improvements of the new algorithms over the other related holistic methods on subsets of TPQs have been demonstrated. The efficiency comes primarily from two sources: the semi-strict matching on P-C edges between two input streams and the various optimization mechanisms (pursued in the buffering technique and the advanced preorder filtering for ordering constraints and positional predicates).

The next chapter concludes the thesis by identifying its main contributions and suggesting some directions for future work.

Chapter 11

Conclusion and Future Work

11.1 Introduction

In the context of XML databases, twig pattern query (TPQ) matching is a core operation in XML query processing because it is how all the matching occurrences of a twig pattern are found in an XML document. The last two decades have seen a growing trend towards TPQ matching. This study contributed to this active area of research by proposing a set of novel twig matching algorithms using the Child Prime Label (CPL) approach. To conclude this thesis, the chapter begins by briefly summarising the thesis and its main contributions. Then, it presents areas for future work.

The remainder of this chapter is organised as follows. Section 11.2 provides a summary of the thesis. A more detailed account of the main contributions of this study is given in Section 11.3. Some directions for future work will be discussed in Section 11.4.

11.2 Thesis Summary

This thesis has investigated problems of twig pattern matching for XML query evaluation. Generally, in XML information retrieval, an XML document and query can be represented by a tree model, and retrieving XML documents can thus be considered as a tree matching problem between the document tree and the query tree. Chapter 2 was devoted to basic concepts of XML and preliminary definitions that are needed for the understanding of the current research.

The purpose of Chapter 3 was to review the literature on XML query processing. It began by introducing the main concepts and techniques exploited in XML retrieval systems. Different aspects of XML query processing have been identified and studied in order to expose inefficiencies in previous work leading to an identification of several limitations in existing approaches. These can be classified into three main categories: inefficiency of the determination of P-C axes in TPQs for algorithms using a partition index, failure to consider the semantics of ordered axes and support positional predicates efficiently and the

performance trade-off between space overhead and optimal enumeration for a combination of preorder and postorder processing for TPQs.

Chapter 4 aimed at highlighting the research problems and motivations, also describing the research methodology adopted in this thesis to ensure a systematic process for carrying out scientific research. In this chapter, the research hypothesis was formulated by proposing an augmentation to the existing labelling schemes in order to improve the efficiency of holistic twig algorithms in terms of running time and memory consumption. Also, it discussed the scope of this study and the potential research objectives. Chapter 5 described the specification and guidelines for implementing the experimental framework in order to evaluate the performance, scalability and efficiency of the new algorithms for processing TPQs.

Chapter 6 presented a new approach that prunes irrelevant elements efficiently for TPQs with P-C axes. It also introduced a novel holistic algorithm which can use the new indexing technique to process TPQs with P-C edges efficiently. Unlike the previous holistic algorithms, TwigStackPrime takes into account the CPL relationships between elements in the streams, therefore it produces fewer path solutions for TPQs with P-C edges. Furthermore, correctness proofs for the filtering strategy using the CPL approach were provided. It has been shown analytically that for TPQs with only A-D edges or P-C relationships related to leaf query nodes, depending on tag partitioning scheme and the CPL approach top-down, TwigStackPrime can guarantee optimal evaluation.

In Chapter 7, the thesis introduced new approaches which consider the ordered axes and sequence operators incorporated into conventional TPQs. In addition, it presented novel techniques which use buffering techniques previously proposed in the literature along with the CPL indexing to process ordered TPQs efficiently. The experimental results have shown that the new holistic ordered twig matching algorithms have superior performance to other related methods with postprocessing in terms of the size of the intermediate storage and query processing time.

The research study, in Chapter 8, presented new approaches that use the CPL approach to improve the filtering phase of bottom-up twig matching algorithms. A novel design of the algorithm uses the level split approach along with the CPL indexing without maintaining stacks in order to process TPQs efficiently. In addition, the research study extended the state-of-the-art bottom-up twig algorithms . Thus, the new methods can skip irrelevant elements efficiently as they utilise the CPL approach devised in Chapter 6. Statistical analysis revealed that the new bottom-up holistic twig matching algorithms significantly outperformed the existing approaches in terms of the number of elements stored in the intermediate storage and query running time.

The research undertaken in this thesis concludes with Chapter 9, in which the discussion of how ordered twig pattern queries (OTPQs) can be evaluated in holistic bottom-up approaches. Two approaches have been proposed to process OTPQs efficiently which are based on filtering checks for ordered constraints and sequence operators from the advanced

preorder filtering functions introduced in Chapter 7. In addition, a novel approach which combines preorder and postorder filtering strategies to identify useful elements in TPQs when positional predicates are involved was presented. The experimental results showed its superiority over the existing twig matching algorithm.

Lastly, the overall evaluation of the experimental design and results was presented in Chapter 10. In this chapter, the experimental results were reconsidered using additional metrics to provide additional results. A summary of the features and limitations of the experiments was discussed in addition to their main findings.

11.3 Main Research Contributions

The contributions of the research presented in this thesis and the related publications [11, 12] can be summarized as follows:

- a new indexing technique which exploits the property of prime numbers to facilitate the determination of P-C axes on XML documents during holistic twig pattern matching has been proposed. It has been demonstrated that the Child Prime Label (CPL) approach can be efficiently incorporated within existing labelling schemes (e.g., range-based encoding).

- an improved approach to reduce the number of overflow cases when creating the CPL index has been introduced.

- a new preorder filtering strategy algorithm has been proposed which is an extension of *getNext()* core function in the classical holistic twig joins algorithm, TwigStack [40], to take advantage of the CPL approach. The new advanced preorder filtering function can efficiently filter out irrelevant elements without violating the *document order* nor consuming additional space overhead. Then, a new top-down holistic algorithm TwigStackPrime has been presented, which is an improvement to TwigStack [40], focusing on reducing the memory consumption and computation overhead of twig pattern matching when P-C edges are involved. Full proofs of correctness for the algorithms necessary to evaluate subsets of TPQs containing P-C and A-D axes have been provided. The behaviour of the CPL approach was explained by analytical results. In particular, holistic algorithms using the CPL indexing technique were shown to be I/O and CPU optimal when a TPQ has only A-D edges or there are P-C edges to connect leaf query nodes. This analysis was confirmed by experimental results on a wide range of real-world, benchmark and artificial datasets.

- new top-down holistic approaches accompanied with optimal implementation algorithms for efficiently processing ordered TPQs have been presented. The new approaches are combinations of the CPL indexing technique and previous methods. The holistic approaches proposed here are the first to consider the semantics of order axes and sequence operators introduced in the XPath specification. Full proofs

of correctness for the algorithms necessary to evaluate subclasses of TPQs with ordering constraints and sequence operators have been provided.

- a set of novel bottom-up holistic twig matching algorithms have been presented which are based on the advanced preorder filtering function introduced in Chapter 6 which has the ability to preserve *the document order*, unlike previous filtering strategies, such as [144, 131], and filter out irrelevant elements when P-C edges are present in TPQs. Full proofs of correctness for the algorithms necessary to evaluate subsets of TPQs containing P-C and A-D axes have been provided.

- a set of novel bottom-up holistic twig matching algorithms for efficiently processing ordered TPQs have been presented.

- a novel bottom-up holistic twig algorithm, OPTwigPrime has been proposed for efficiently processing ordered TPQs with positional predicates. Up to now, OPTwigPrime is the first holistic twig matching algorithm designed for TPQs which may contain ordering constraints and positional predicates, including a number of original supporting techniques. Full proofs of correctness for the algorithm necessary to evaluate subsets of TPQs with ordered axes, sequence operators and positional predicates have been provided.

- lastly, the study has provided an empirical proof of improvements of the holistic algorithms proposed, based on the CPL approach, over other related methods from the literature.

11.4 Future Work

The empirical findings in this study provide a new understanding of the link between the information contained in XML labels and the efficiency of XML twig matching algorithms in terms of processing time capabilities and memory consumption. The present study was designed to determine the effect of the CPL approaches on various common subsets of TPQs, however there is plenty of scope for further investigation. Some directions for future work are:

Further experimental investigations are needed to explore the effects of the CPL approach by using a different labelling scheme such as E2Dewey [125] instead of the range-based encoding scheme used in this study. Such an advanced labelling scheme that can fit in the framework of the CPL approach for more effectively skipping non-contributing elements while reading streams corresponding to branching and leaf query nodes.

Since the study was limited to holistic algorithms which do not use structural summaries, it was not possible to evaluate the performance of the CPL approaches with

methods which combine structural summaries and node labelling schemes. Further research needs to examine more closely the links between the CPL indexing technique and different streaming schemes rather than the tag streaming scheme. It would be interesting to assess the effects of the CPL indexing technique to classify elements based on their forward bi-simulation in order to reduce the number of elements scanned (reducing I/O cost). This could be expected to overcome some limitations associated with the prefix path scheme (PPS) which involved a large number of streams and induced inefficiency [130, 24]. Figure 11.1 illustrates a potential partitioning scheme using the CPL approach.

Example 11.1. *Consider the XML tree T_1 of Figure 11.1a and the TPQ = //a[/x]/y. If the tag streaming is used as in Figure 11.1b, five elements corresponding to the query node a have to be scanned, namely $\{a_1, a_2, a_3, a_4, a_5\}$ even though only $\{a_1, a_5\}$ contribute to the final result. On the other hand, two elements, namely $\{a_1, a_5\}$ are scanned when using the CPL streaming scheme since the stream a^{35} satisfies the CPL relationship with respect to the query node x and y as shown in in Figure 11.1e.*

Another suggestion for future work is to combine the algorithms proposed with previous orthogonal approaches such as useless elements skipping [114, 114, 77], refined partitioning [21, 50], virtual streams [125] and content search [235, 236].

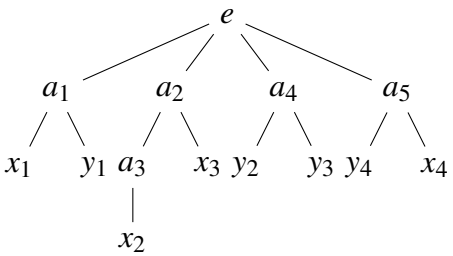
The current study has not yet addressed the type of XML queries that involve reverse axes and wildcard ("*") node. Incorporating the support for reverse axes (i.e., "\" and "\\") and wildcard nodes into the CPL approaches presented in this thesis is another interesting topic to study in the future. Besides, further work needs to be done to investigate the behaviour of the CPL approach when processing boolean expressions and supporting the GTP semantics.

Lastly, it would be interesting to see an elegant algorithm which can guarantee linear CPU and I/O complexities of the output enumeration with respect to the output size for TPQs with ordered axes and positional predicates, and does not perform multiple scans of input streams.

11.5 Finally

This research has investigated a wide range of XML query evaluation approaches leading to the identification of several limitations which motivated the introduction of the CPL approach and numerous original supporting techniques. The CPL is a novel indexing technique that provides an excellent solution in terms of query efficiency and ability. However, there is still room for improvement.

In this chapter, a summary of the thesis was presented followed by a list of its main contributions and some directions for suture work. All in all, the contributions of this study have been to confirm that the CPL is a promising alternative approach for twig pattern matching of XML query processing and therefore, further research in this direction may be worth pursuing.



(a) an XML tree T_1

e:$\{e\}$
a:$\{a_1, a_2, a_3, a_4, a_5\}$
x:$\{x_1, x_2, x_3, x_4\}$
y:$\{y_1, y_2, y_3, y_4\}$

(b) tag streams

e^1:$\{e\}$
a^2:$\{a_1, a_2, a_4, a_5\}$
a^3:$\{a_3\}$
x^3:$\{x_1, x_3, x_4\}$
x^4:$\{x_2\}$
y^3:$\{y_1, y_2, y_3, y_4\}$

(c) tag+level streams

/e:$\{e\}$
/e/a:$\{a_1, a_2, a_4, a_5\}$
/e/a/a:$\{a_3\}$
/e/a/x:$\{x_1, x_3, x_4\}$
/e/a/a/x:$\{x_2\}$
/e/a/y:$\{y_1, y_2, y_3, y_4\}$

(d) prefix path streams

e^3:$\{e\}$
a^{35}:$\{a_1, a_5\}$
a^5:$\{a_2, a_3\}$
a^7:$\{a_4\}$
x^1:$\{x_1, x_2, x_3, x_4\}$
y^1:$\{y_1, y_2, y_3, y_4\}$

(e) CPL streams

Tagname	Key
e	2
a	3
x	5
y	7

(f) tag indexing

Figure 11.1: Illustration to the CPL partitioning scheme.

References

[1] Santa Agreste, Pasquale De Meo, Emilio Ferrara, and Domenico Ursino. XML Matchers: Approaches and challenges. *Knowledge-Based Systems*, 66(0):190–209, 2014. doi: http://dx.doi.org/10.1016/j.knosys.2014.04.044.

[2] Jinhyun Ahn, Dong Hyuk Im, Taewhi Lee, and Hong Gee Kim. Parallel prime number labeling of large XML data using MapReduce. In *6th International Conference on IT Convergence and Security, ICITCS 2016*, pages 1–2, 2016. ISBN 9781509037643. doi: 10.1109/ICITCS.2016.7740360.

[3] Mohammed Al-Badawi. *A Performance Evaluation of a New Bitmap-based XML Processing Approach*. PhD thesis, The University of Sheffield, 2010.

[4] Hamdi A Al-jamimi, Ahmed Barradah, and Salahadin Mohammed. Siblings Labeling Scheme for Updating XML Trees Dynamically. In *4th International Conference on Computer Engineering and Technology (ICCET 2012)*, volume 40, pages 21–25, 2012.

[5] S Al-Khalifa, H V Jagadish, N Koudas, J M Patel, D Srivastava, and Wu Yuqing. Structural joins: a primitive for efficient XML query pattern matching. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 141–152, 2002. ISBN 1063-6382. doi: 10.1109/icde.2002.994704.

[6] N S Alghamdi, W Rahayu, and E Pardede. Object-Based Semantic Partitioning for XML Twig Query Optimization. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 846–853, 2013. ISBN 1550-445X. doi: 10.1109/aina.2013.74.

[7] Norah Saleh Alghamdi, Wenny Rahayu, and Eric Pardede. Semantic-based structural and content indexing for the efficient retrieval of queries over large XML data repositories. *Future Generation Computer Systems*, 2014. ISSN 0167739X. doi: 10.1016/j.future.2014.02.010.

[8] Norah Saleh Alghamdi, Wenny Rahayu, and Eric Pardede. Efficient Processing of Queries over Recursive XML Data. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, number June, pages 134–142, 2015. ISBN 978-1-4799-7905-9. doi: 10.1109/AINA.2015.177.

[9] S Alireza Aghili, Li Hua-Gang, Divyakant Agrawal, and Amr El Abbadi. TWIX: twig structure and content matching of selective queries using. *InfoScale '06: Proceedings of the 1st international conference on*, page 42, 2006. doi: 10.1145/1146847.1146889.

[10] Alaa Almelibari. *Labelling Dynamic XML Documents: A GroupBased Approach*. PhD thesis, 2015.

[11] Shtwai Alsubai and Siobhán North. A Prime Number Approach to Matching an XML Twig Pattern including Parent-Child Edges. In *The 13th International Conference on Web Information Systems and Technologies (WEBIST 2017)*, pages 204–211, Porto, 2017. SCITEPRESS – Science and Technology Publications, Lda.

[12] Shtwai Alsubai and Siobhán North. TwigStackPrime : A Novel Twig Join Algorithm Based on Prime Numbers. *Lecture Note Business Information Processing, Revised Selected Papers WEBIST 2017, Springer (in press)*, 2018.

[13] T Amagasa, M Seino, and H Kitagawa. Energy-Efficient XML Stream Processing through Element-Skipping Parsing. In *Database and Expert Systems Applications (DEXA), 2013 24th International Workshop on*, pages 254–258, 2013. ISBN 1529-4188. doi: 10.1109/dexa.2013.34.

[14] Jose Nelson Amaral, Michael Buro, Renee Elio, Jim Hoover, Ioanis Nikolaidis, Mohammad Salavatipour, Lorna Stewart, and Ken Wong. About Computing Science Research Methodology, 2011. URL https://webdocs.cs.ualberta.ca/{~}c603/readings/research-methods.pdf.

[15] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of tree pattern queries. *ACM SIGMOD Record*, 30:497–508, 2001. ISSN 01635808. doi: 10.1145/376284.375730.

[16] P Apparao, R Iyer, R Morin, Nayak Naren, M Bhat, D Halliwell, and W Steinberg. Architectural characterization of an XML-centric commercial server workload. In *Parallel Processing, 2004. ICPP 2004. International Conference on*, pages 292–300 vol.1, 2004. ISBN 0190-3918. doi: 10.1109/icpp.2004.1327935.

[17] Andrei Arion, Angela Bonifati, Ioana Manolescu, and Andrea Pugliese. Path summaries and path partitioning in modern XML databases. *World Wide Web*, 11 (1):117–151, 2008. ISSN 1386145X. doi: 10.1007/s11280-007-0036-7.

[18] Radim Bača and Michal Krátký. On the Efficiency of a Prefix Path Holistic. In *Database and XML Technologies: 6th International XML Database Symposium, XSym 2009, Lyon, France, August 24, 2009. Proceedings*, number 201, pages 25–32, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03555-5. doi: 10.1007/978-3-642-03555-5_3. URL http://dx.doi.org/10.1007/978-3-642-03555-5{_}3.

[19] Radim Bača and Michal Krátký. TJDewey - On the Efficient Path Labeling Scheme Holistic Approach. In Lei Chen, Chengfei Liu, Qing Liu, and Ke Deng, editors, *Database Systems for Advanced Applications: DASFAA 2009 International Workshops: BenchmarX, MCIS, WDPP, PPDA, MBC, PhD, Brisbane, Australia, April 20 - 23, 2009*, number 201, pages 6–20, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-04205-8. doi: 10.1007/978-3-642-04205-8_3.

[20] Radim Bača and Michal Krátký. XML query processing. *Proceedings of the 16th International Database Engineering & Applications Sysmposium on - IDEAS '12*, pages 8–13, 2012. doi: 10.1145/2351476.2351478.

[21] Radim Bača, Michal Krátký, and Václav Snášel. On the Efficient Search of an XML Twig Query in Large DataGuide Trees. In *Proceedings of the 2008 International Symposium on Database Engineering & Applications*, IDEAS '08, pages 149–158, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-188-0. doi: 10.1145/1451940.1451962.

[22] Radim Bača, Michal Krátký, Tok Wang Ling, and Jiaheng Lu. Optimal and efficient generalized twig pattern processing: a combination of preorder and postorder filterings. *The VLDB Journal*, 22(3):369–393, oct 2012. ISSN 1066-8888. doi: 10.1007/s00778-012-0295-5.

[23] Radim Bača, Petr Lukáš, and Michal Krátký. Cost-based Holistic Twig Joins. *Information Systems*, 52(C):21–33, aug 2015. ISSN 0306-4379. doi: 10.1016/j.is.2015.03.004.

[24] Radim Bača, Michal Krátký, Irena Holubová, Martin Nečaský, Tomáš Skopal, Martin Svoboda, and Sherif Sakr. Structural XML Query Processing. *ACM Computing Surveys*, 50(5):1–41, 2017. ISSN 03600300. doi: 10.1145/3095798.

[25] Zhifeng Bao, Tok Wang Ling, Jiaheng Lu, and Bo Chen. SemanticTwig : A Semantic Approach to Optimize XML Query Processing. In *Database Systems for Advanced Applications 13th International Conference, DASFAA 2008, New Delhi, India, March 19-21, 2008. Proceedings*, pages 282–298, 2008.

[26] Zhifeng Bao, Tok Wang Ling, Bo Chen, and Jiaheng Lu. Effective XML keyword search with relevance oriented ranking. In *Proceedings - International Conference on Data Engineering*, number 3, pages 517–528, 2009. ISBN 9780769535456. doi: 10.1109/ICDE.2009.16.

[27] Charles Barton, Philippe Charles, Deepak Goyal, and Mukund Raghavachari. Streaming XPath Processing with Forward and Backward Axes. In *In Proceedings of the 19th International Conference on Data Engineering ICDE*, pages 455—-466, 2003.

[28] Yoav Benjamini. TEACHER ' S CORNER In: Opening the Box of a Boxplot. *The American Statistician*, 42(4):257–262, 1988.

[29] Mikael Berndtsson, Jörgen Hansson, B. Olsson, and Björn Lundell. *Thesis Projects: A Guide for Students in Computer Science and Information Systems*. Number 1. Springer-Verlag London, 2 edition, 2008. ISBN 978-1-84800-009-4. doi: 10.1007/978-1-84800-009-4.

[30] Nicole Bidoit, Dario Colazzo, Noor Malla, Federico Ulliana, Maurizio Nolè, and Carlo Sartiani. Processing XML queries and updates on map/reduce clusters. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 745–748, Genoa, Italy, 2013. ACM. doi: 10.1145/2452376.2452470.

[31] Philip Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1-3):217–239, 2005. doi: 10.1016/j.tcs.2004.12.030.

[32] The DBLP Advisory Board. DBLP. URL http://dblp.uni-trier.de/xml/.

[33] Timo Böhme and Erhard Rahm. *XMach-1: A Multi-User Benchmark for XML Data Management.*, pages 264–273. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-642-56687-5. doi: 10.1007/978-3-642-56687-5_20.

[34] Angela Bonifati. XPathMark: An XPath Benchmark for the XMark Generated Data. *Database and XML Technologies*, 3671(July 2015), 2005. doi: 10.1007/11547273. URL http://www.springerlink.com/content/19l0fpmekdlcdbjw/.

[35] Stephane Bressan, Gillian Dobbie, Zoe Lacroix, Mong-Li Lee, Ying Guang Li, Ullas Nambiar, and Bimlesh Wadhwa. X007: Applying 007 Benchmark To Xml Query Processing Tool. In *Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management*, pages 167–174, Atlanta, Georgia, USA, 2001.

[36] David Brownell. *SAX2 Processing XML Efficiently with Java*. O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472., 2004. ISBN 0596002378.

[37] Achim D. Brucker and Jacques Julliand. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing Verification and Reliability*, 24(8):591–592, 2014. ISSN 10991689. doi: 10.1002/stvr.

[38] N Bruno, L Gravano, N Koudas, and D Srivastava. Navigation- vs. index-based XML multi-query processing. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 139–150, 2003. doi: 10.1109/icde.2003.1260788.

[39] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Technical Report - Holistic twig joins: optimal XML pattern matching. Technical report, 2002.

[40] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321, Madison, Wisconsin, 2002. ACM. doi: 10.1145/564691.564727.

[41] Michael J Carey, David J DeWitt, Chander Kant, and Jeffrey F Naughton. A Status Report on the OO7 OODBMS Benchmarking Effort. *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, pages 414–426, 1994. ISSN 03621340. doi: 10.1145/191080.191147. URL http://doi.acm.org/10.1145/191080.191147.

[42] Balder Ten Cate and Luc Segoufin. Transitive Closure Logic, Nested TreeWalking Automata, and XPath. *Journal of the ACM*, 57(3):1–41, 2010. ISSN 00045411. doi: 10.1145/1376916.1376952.

[43] B Cautis and E Kharlamov. Answering queries using views over probabilistic XML: complexity and tractability. *Proceedings of the VLDB Endowment*, 5:1148–1159, 2012. ISSN 21508097. URL http://dl.acm.org/citation.cfm?id=2350235.

[44] Georgetown University Medical Center and University of Delaware. The Universal Protein Resource (UniProt), 2007. URL http://pir.georgetown.edu/.

[45] Stefano Ceri, Piero Fraternali, and Stefano Paraboschi. XML: Current Developments and Future Challenges for the Database Community. In Carlo Zaniolo, Peter C Lockemann, Marc H Scholl, and Torsten Grust, editors, *Advances in Database Technology EDBT: 7th International Conference on Extending Database Technology Konstanz, Germany*, pages 3–17, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-46439-6. doi: 10.1007/3-540-46439-5_1.

[46] Don Chamberlin, Harry Road, San Jose, Jonathan Robie, and Daniela Florescu. Quilt : An XML Query Language for Heterogeneous Data Sources. *Language*, pages 1–10, 2000.

[47] Cy Chan, Wenfei Fan, and Y Zeng. Taming XPath queries by minimizing wildcard steps. *Vldb*, pages 156–167, 2004. URL http://portal.acm.org/citation.cfm?id=1316689.1316705.

[48] Tao-Ku Chang and Gwan-Hwan Hwang. Developing an efficient query system for encrypted XML documents. *Journal of Systems and Software*, 84(8):1292–1305, 2011. doi: http://dx.doi.org/10.1016/j.jss.2011.04.012.

[49] Surajit Chaudhuri, Zhiyuan Chen, Kyuseok Shim, and Yuqing Wu. Storing XML (with XSD) in SQL databases: Interplay of logical and physical designs. *IEEE Transactions on Knowledge and Data Engineering*, 17(12):1595–1609, 2005. ISSN 10414347. doi: 10.1109/TKDE.2005.204.

[50] Bo Chen, Tok Wang Ling, M Tamer Ozsu, and Zhenzhou Zhu. On Label Stream Partition for Efficient Holistic Twig Join. In *12th International Conference on Database Systems for Advanced Applications, DASFAA*, pages 807–818, Bangkok, Thailand, 2007. ISBN 9783540717027.

[51] Liang Jeff Chen and Yannis Papakonstantinou. Supporting top-K keyword search in XML databases. *Proceedings - International Conference on Data Engineering*, pages 689–700, 2010. ISSN 10844627. doi: 10.1109/ICDE.2010.5447818.

[52] Qun Chen, Andrew Lim, and Kian Win Ong. D(k)-index: an adaptive structural summary for graph-structured data. *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 134–144, 2003. ISSN 07308078. doi: 10.1145/872757.872776.

[53] Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K Selçuk Candan. Twig2Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proceedings of the 32nd international conference on Very large data bases*, pages 283–294, Seoul, Korea, 2006. VLDB Endowment.

[54] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques. In *ACM SIGMOD international conference on Management of data*, pages 455–466, 2005. ISBN 1595930604. doi: 10.1145/1066157.1066209.

[55] Yangjun Chen and Donovan Cooke. Evaluation of Twig Pattern Queries Based on Ordered Tree matching. In *IEEE International Conference on Signal Image Technology and Internet Based Systems Evaluation*, pages 3–10, 2008. ISBN 9780769534930. doi: 10.1109/SITIS.2008.32.

[56] Yi Chen, Susan B Davidson, and Yifeng Zheng. BLAS: An Efficient XPath Processing System. In *ACM SIGMOD Int. Conf. on Management of data*, pages 47–58, 2004. ISBN 1-58113-859-8. doi: http://doi.acm.org/10.1145/1007568.1007577.

[57] Yi Chen, George A Mihaila, Susan B Davidson, and Sriram Padmanabhan. Efficient Path Query Processing on Encoded XML. In *International Workshop on High Performance XML Processing*, 2004.

[58] Zhimin Chen, H V Jagadish, Laks V S Lakshmanan, and Stelios Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Proceedings of the 29th international conference on Very large data bases*, pages 237–248, 2003. ISBN 0127224424. URL http://www.vldb.org/conf/2003/papers/S08P03.pdf.

[59] Zhiyuan Chen, Johannes Gehrke, Flip Korn, Nick Koudas, Jayavel Shanmugasundaram, and Divesh Srivastava. Index structures for matching XML twigs using relational query processors. *Data and Knowledge Engineering*, 60:283–302, 2007. ISSN 0169023X. doi: 10.1016/j.datak.2006.03.003.

[60] Su Cheng Haw, Samini Subramaniam, Wei Siang Lim, and Fang Fang Chua. Hybridation of Labeling Schemes for Efficient Dynamic Updates. *Indonesian Journal of Electrical Engineering and Computer Science*, 4(1):184, 2016. ISSN 2502-4760. doi: 10.11591/ijeecs.v4.i1.pp184-194.

[61] B Choi, M Mahoui, and D Wood. On the optimality of holistic algorithms for twig queries. In *Database and Expert Systems Applications 14th International Conference, DEXA 2003, Prague, Czech Republic, September 1-5, 2003, Proceedings*, pages 28–37, 2003.

[62] Kajal T. Claypool, Vaishali Hegde, and Naiyana Tansalarak. QMatch - A hybrid match algorithm for XML schemas. In *Proceedings - International Workshop on Biomedical Data Engineering, BMDE2005*, volume 2005, 2005. ISBN 0769526578. doi: 10.1109/ICDE.2005.272.

[63] J W Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications, Inc, 4th edition, 2013. ISBN 9781452274614. doi: 10.1007/s13398-014-0173-7.2.

[64] Harold D Delaney and Andrfis Vargha. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[65] Amy B Dellinger and Nancy L Leech. Toward a Unified Validation Framework in Mixed Methods Research. *Journal of Mixed Methods Research*, 1(4):309–332, 2007. ISSN 1558-6898. doi: 10.1177/1558689807306147. URL http://mmr.sagepub.com/content/1/4/309.abstract.

[66] Dabin Ding, Dunren Che, and Wen-Chi Hou. A Direct Approach to Holistic Boolean-Twig Pattern Evaluation. In *23rd International Conference Database and Expert Systems Applications (DEXA)*, pages 342–356, 2012.

[67] Dabin Ding, Dunren Che, Fei Cao, and Wen-Chi Hou. *A Practical Approach to Holistic B-Twig Pattern Matching for Efficient XML Query Processing*, pages 165–174. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-39467-6. doi: 10.1007/978-3-642-39467-6_16.

[68] Che Dunren, Ling Tok Wang, and Hou Wen-Chi. Holistic Boolean-Twig Pattern Matching for Efficient XML Query Processing. *Knowledge and Data Engineering, IEEE Transactions on*, 24(11):2008–2024, 2012. doi: 10.1109/tkde.2011.128.

[69] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting Empirical Methods for Software Engineering Research. *Guide to Advanced Empirical Software Engineering*, pages 285–311, 2008. ISSN <null>. doi: 10.1007/978-1-84800-044-5_11. URL http://link.springer.com/chapter/10.1007/978-1-84800-044-5{_}11.

[70] Li Fajin, Liao Husheng, and Gao Hongyu. Twig Pattern Matching with Positional Predicates in XML Queries. In *Web Information System and Application Conference (WISA), 2013 10th*, pages 113–118, 2013. doi: 10.1109/wisa.2013.30. URL http://ieeexplore.ieee.org/ielx7/6777803/6778588/06778621.pdf?tp={&}arnumber=6778621{&}isnumber=6778588.

[71] David C Fallside and Priscilla Walmsley. XML Schema Part 0: Primer Second Edition, 2004. URL http://www.w3.org/TR/xmlschema-0/.

[72] Norah Saleh Farooqi. *Applying Dynamic Trust Based Access Control to Improve XML Databases Security* . Phd thesis, The University of Sheffield, 2013.

[73] T Fiebig, S Helmer, C.-C. Kanne, G Moerkotte, J Neumann, R Schiele, and T Westmann. Anatomy of a Native XML Base Management System. *The VLDB Journal*, 11(4):292–314, dec 2002. ISSN 1066-8888. doi: 10.1007/s00778-002-0080-y. URL http://dx.doi.org/10.1007/s00778-002-0080-y.

[74] Andy Field, Jeremy Miles, and Zoë Field. *Discovering Statistics Using IBM SPSS Statistics*, volume 81. SAGE Publications Ltd, 3 edition, 2013. ISBN "9781847879066". doi: 10.1111/insr.12011_21.

[75] Peter M. Fischer, Aayush Garg, and Kyumars Sheykh Esmaili. Extending XQuery with a pattern matching facility. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6309 LNCS(1):48–57, 2010. ISSN 03029743. doi: 10.1007/978-3-642-15684-7_5.

[76] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin, Special Issue on 1060*, 22(3):27–34, 1999. URL http://dsl.serc.iisc.ernet.in/{~}course/TIDS/papers/daniela.pdf.

[77] Marcus Fontoura, Vanja Josifovski, Eugene Shekita, and Beverly Yang. Optimizing Cursor Movement in Holistic Twig Joins. In *Proceeding CIKM '05 Proceedings of the 14th ACM international conference on Information and knowledge management Pages 784-791*, pages 784–791, 2005. ISBN 1595931406. doi: 10.1145/1099554.1099741.

[78] Massimo Franceschet. XPathMark : an XPath Benchmark for the XMark Generated Data. In *Third International XML Database Symposium XSym, Trondheim, Norway.*, 2005.

[79] Massimo Franceschet, Donatella Gubiani, Angelo Montanari, and Carla Piazza. A graph-theoretic approach to map conceptual designs to XML schemas. *ACM Trans. Database Syst.*, 38(1):1–44, 2013. doi: 10.1145/2445583.2445589.

[80] Catherine O Fritz, Peter E Morris, and Jennifer J Richler. Effect size estimates: current use, calculations, and interpretation. *Journal of experimental psychology. General*, 141(1):2–18, 2012. ISSN 1939-2222. doi: 10.1037/a0024338. URL http://www.ncbi.nlm.nih.gov/pubmed/21823805.

[81] Gou Gang and Rada Chirkova. Efficiently querying large XML data repositories: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1381–1403, 2007. ISSN 10414347. doi: 10.1109/TKDE.2007.1060.

[82] Salvador García, Alberto Fernández, Julián Luengo, and Francisco Herrera. Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. *Information Sciences*, 180(10):2044–2064, 2010. ISSN 00200255. doi: 10.1016/j.ins.2009.12.010.

[83] Haris Georgiadis, Minas Charalambides, and Vasilis Vassalos. A Query Optimization Assistant For Xpath. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 550–553, 2011. ISBN 9781450305280. doi: 10.1145/1951365.1951438.

[84] T A Ghaleb and S Mohammed. Novel scheme for labeling XML trees based on bits-masking and logical matching. In *Computer and Information Technology (WCCIT), 2013 World Congress on*, pages 1–5, 2013. doi: 10.1109/wccit.2013.6618715.

[85] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *proceedings of the international conference on Very Large Data Bases*, pages 436–445, Athens, Greece, 1997. ISBN 3060296103. doi: 10.1.1.15.9610.

[86] Roy Goldman, Jason McHugh, and Jennifer Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *ACM SIGMOD Workshop on The Web and Databases (WebDB 1999)*, pages 25–30, 1999.

[87] Nils Grimsmo and Truls A Bjørklund. XLeaf : Twig Evaluation with Skipping Loop Joins and Virtual Nodes. In *Second International Conference on Advances in Databases, Knowledge, and Data Applications*, 2010. ISBN 9780769539812. doi: 10.1109/DBKDA.2010.8.

[88] Nils Grimsmo and Truls A Bjørklund. Towards Unifying Advances in Twig Join Algorithms. In *Proceedings of the Twenty-First Australasian Conference on Database Technologies*, volume 104 of *ADC '10*, pages 57–66, Darlinghurst, Australia, Australia, 2010. Australian Computer Society, Inc. ISBN 978-1-920682-85-9.

[89] Nils Grimsmo, Truls Amundsen Bjørklund, and Magnus Lie Hetland. Fast optimal twig joins. *Proceedings of the VLDB Endowment*, 3(1-2):894–905, sep 2010. ISSN 21508097. doi: 10.14778/1920841.1920955.

[90] Torsten Grust. Accelerating XPath Location Steps. *Proceedings of the 2002 ACM SIGMOD international conference on Management of data - SIGMOD '02*, page 109, 2002. ISSN 07308078. doi: 10.1145/564704.564705. URL http://portal.acm.org/citation.cfm?doid=564691.564705.

[91] Torsten Grust, Maurice van Keulen, and Jens Thilo Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. *Proceedings of the 29th International Conference on Very Large Databases*, pages 524—-535, 2003.

[92] Liu Guiquan, Yao Meiling, Wang Desheng, and Chen Enhong. A novel three-phase XML twig pattern matching algorithm based on version tree. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on*, volume 3, pages 1678–1688, 2011. doi: 10.1109/fskd.2011.6019809.

[93] Marouane Hachicha and Jérôme Darmont. A survey of XML tree patterns. In *IEEE Transactions on Knowledge and Data Engineering*, volume 25, pages 29–46, 2013. doi: 10.1109/TKDE.2011.209.

[94] Theo Härder, Michael Haustein, Christian Mathis, and Markus Wagner. Node labeling schemes for dynamic XML documents reconsidered. *Data and Knowledge Engineering*, 60(1):126–149, 2007. ISSN 0169023X. doi: 10.1016/j.datak.2005.11.008.

[95] Su-Cheng Haw and Chien-Sing Lee. TwigINLAB : A Decomposition-Matching-Merging Approach To Improving XML Query Processing. *American Journal of Applied Sciences*, 5(9):1199–1205, 2008.

[96] Su Cheng Haw and Chien Sing Lee. TwigX-Guide: An efficient twig pattern matching system extending dataguide indexing and region encoding labeling. *Journal of Information Science and Engineering*, 2009. ISSN 10162364.

[97] Su Cheng Haw and G. S V Radha Krishna Rao. A comparative study and benchmarking on XML parsers. In *International Conference on Advanced Communication Technology, ICACT*, volume 1, pages 321–325, 2007. ISBN 8955191316. doi: 10.1109/ICACT.2007.358364.

[98] Su-Cheng Cheng Haw and Chien-Sing Sing Lee. Data storage practices and query processing in XML databases: A survey. *Knowledge-Based Systems*, 24(8):1317–1340, 2011. ISSN 09507051. doi: 10.1016/j.knosys.2011.06.006.

[99] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105, 2004. ISSN 02767783. doi: 10.2307/25148625. URL http://dblp.uni-trier.de/rec/bibtex/journals/misq/HevnerMPR04.

[100] Hilary J Holz, Anne Applin, Donald Joyce, Helen Purchase, Catherine Reed, and Bruria Haberman. Research Methods in Computing : What are they , and how should we teach them ? *Information Systems*, pages 96–114, 2006. ISSN 00978418. doi: 10.1145/1189136.1189180.

[101] Mingsheng Hong, Alan J Demers, Johannes E Gehrke, Christoph Koch, Mirek Riedewald, and Walker M White. Massively multi-query join processing in publish/subscribe systems, 2007.

[102] Wen-Chiao Hsu and I En Liao. CIS-X: A compacted indexing scheme for efficient query evaluation of XML documents. *Information Sciences*, 241(0):195–211, 2013. doi: http://dx.doi.org/10.1016/j.ins.2013.03.055.

[103] Gongzhu Hu and Chunxia Tang. Indexing XML Data for Path Expression Queries. In *First International Conference, SERA: International Conference on Software Engineering Research and Applications*, pages 332–348, 2003.

[104] Sascha Hunold and Jesper Larsson Träff. On the state and importance of reproducible experimental research in parallel computing. *arXiv preprint arXiv:1308.3648*, pages 1–15, 2013.

[105] Sayyed Kamyar Izadi, Theo Härder, and Mostafa S. Haghjoo. S3: Evaluation of tree-pattern XML queries supported by structural summaries. *Data and Knowledge Engineering*, 68(1):126–145, 2009. ISSN 0169023X. doi: 10.1016/j.datak.2008.09.001. URL http://dx.doi.org/10.1016/j.datak.2008.09.001.

[106] Sayyed Kamyar Izadi, Mostafa S. Haghjoo, and Theo Härder. S3: Processing tree-pattern XML queries with all logical operators. *Data and Knowledge Engineering*, 72:31–62, 2012. ISSN 0169023X. doi: 10.1016/j.datak.2011.09.003.

[107] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V S Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. Timber: A native XML database. *VLDB Journal*, 11:274–291, 2002. ISSN 10668888. doi: 10.1007/s00778-002-0081-x.

[108] H V Jagadish, Laks V S Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A Tree Algebra for XML. In Giorgio Ghelli and Gösta Grahne, editors, *Database Programming Languages: 8th International Workshop, DBPL 2001 Frascati, Italy, September 8–10, 2001 Revised Papers*, pages 149–164, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-46093-0. doi: 10.1007/3-540-46093-4_9.

[109] Ahmed Jedidi, Olfa Arfaoui, and Minyar Sassi-Hidri. Indexing Compressed XML Documents. In Zhifeng Bao, Yunjun Gao, Yu Gu, Longjiang Guo, Yingshu Li, Jiaheng Lu, Zujie Ren, Chaokun Wang, and Xiao Zhang, editors, *Web-Age Information Management*, volume 7419, chapter 31, pages 319–328. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-33049-0. doi: 10.1007/978-3-642-33050-6_31. URL http://dx.doi.org/10.1007/978-3-642-33050-6{_}31.

[110] Lu Jiaheng. *EFFICIENT PROCESSING OF XML TWIG*. Phd thesis, National University Of Singapore, 2006.

[111] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jx Yu. Holistic twig joins on indexed XML documents. *Proceedings of the 29th international conference on Very large data bases*, Volume 29:273 – 284, 2003.

[112] Haifeng Jiang, Hongjun Lu, and Wei Wang. Efficient Processing of XML Twig Queries with OR-Predicates. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 59—-70, Paris, France, 2004. ACM.

[113] Haifeng Jiang, Hongjun Lu, and Wei Wang. Efficient processing of XML twig queries with OR-predicates. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 59–70, Paris, France, 2004. ACM. doi: 10.1145/1007568.1007578.

[114] Haifeng Jiang Haifeng Jiang, Hongjun Lu Hongjun Lu, Wei Wang Wei Wang, and B.C. Ooi. XR-tree: indexing XML data for efficient structural joins. *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, pages 253–264, 2003. ISSN 1063-6382. doi: 10.1109/ICDE.2003.1260797.

[115] Zhewei Jiang, Cheng Luo, and Wen-Chi Hou. An efficient one-phase holistic twig join algorithm for XML data. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 786–787, Arlington, Virginia, USA, 2006. ACM. doi: 10.1145/1183614.1183730.

[116] Zhewei Jiang, Cheng Luo, Wen-Chi Hou, Qiang Zhu, and Dunren Che. Efficient Processing of XML Twig Pattern : A Novel One-Phase Holistic Solution. In *Database and Expert Systems Applications: 18th International Conference, DEXA 2007, Regensburg, Germany, September 3-7, 2007. Proceedings*, pages 87–97, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 9783540744672. doi: 10.1007/978-3-540-74469-6_10.

[117] Tang Jie, Liu Shaoshan, Liu Chen, Gu ZhiMin, and J L Gaudiot. Acceleration of XML Parsing through Prefetching. *Computers, IEEE Transactions on*, 62(8): 1616–1628, 2013. doi: 10.1109/tc.2012.88.

[118] Jiang Jinhua, Chen Gang, Shou Lidan, and Chen Ke. OTJFast: Processing Ordered XML Twig Join Fast. In *the 3rd IEEE Asia-Pacific Services Computing Conference, APSCC*, number 60603044, pages 1289–1294, 2008. ISBN 9780769534732. doi: 10.1109/APSCC.2008.15.

[119] Raghav Kaushik, Philip Bohannon, Jeffrey F Naughton, and Henry F Korth. Covering indexes for branching path queries. *Proceedings of the 2002 ACM SIGMOD international conference on Management of data - SIGMOD '02*, page 133, 2002. ISSN 07308078. doi: 10.1145/564704.564707. URL http://portal.acm.org/citation.cfm?doid=564691.564707.

[120] Kay Whatley. XML basics for new users, 2009. URL http://www.ibm.com/developerworks/library/x-newxml/.

[121] M Khabbaz, D Assi, R Alhajj, and M Hammad. Parse tree based approach for processing XML streams. In *Information Reuse and Integration (IRI), 2013 IEEE 14th International Conference on*, pages 546–553, 2013. doi: 10.1109/iri.2013.6642517.

[122] Laks V S Lakshmanan, Ganesh Ramesh, Hui Wang, and Zheng (Jessica) Zhao. On Testing Satisfiability of Tree Pattern Queries. *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 120–131, 2004.

[123] Prashant R. Lambole and Prashant N. Chatur. A review on XML keyword query processing. In *IEEE International Conference on Innovative Mechanisms for Industry Applications, ICIMIA 2017 - Proceedings*, number Icimia, pages 238–241, 2017. ISBN 9781509059607. doi: 10.1109/ICIMIA.2017.7975610.

[124] Ki Hoon Lee, Kyu Young Whang, Wook Shin Han, and Min Soo Kim. Structural consistency: Enabling XML keyword search to eliminate spurious results consistently. *VLDB Journal*, 19(4):503–529, 2010. ISSN 10668888. doi: 10.1007/s00778-009-0177-7.

[125] Sangkeun Lee, Byung Gul Ryu, and Kun Lung Wu. Examining the impact of data-access cost on XML twig pattern matching. *Information Sciences*, 203:24–43, 2012. ISSN 00200255. doi: 10.1016/j.ins.2012.03.011.

[126] Changqing Li and Tok Wang Ling. *Advanced Applications and Structures in XML processing: Label streams, semantics utilization and data query technologies*. Hershey, PA: IGI Global, 2010. ISBN 9781615207275. doi: 10.4018/978-1-61520-727-5.

[127] Dong Li, Xiuyu Lu, Xifeng Huang, and Wenhao Chen. Structural Join in the 'XSQS' Native XML Database. *JOURNAL OF SOFTWARE*, 8(1), 2013.

[128] Dongyang Li and Chunping Li. TRACK: A novel XML join algorithm for efficient processing twig queries. In *Conferences in Research and Practice in Information Technology Series*, volume 75, pages 137–143, 2008.

[129] Fei Li, Hongzhi Wang, Liang Hao, Jianzhong Li, and Hong Gao. Approximate joins for XML at label level. *Information Sciences*, 282(0):237–249, 2014. doi: http://dx.doi.org/10.1016/j.ins.2014.06.007.

[130] Guoliang Li, Jianhua Feng, Yong Zhang, and Lizhu Zhou. Efficient Holistic Twig Joins in Leaf-to-Root Combining with Root-to-Leaf Way. In Ramamohanarao Kotagiri, P Radha Krishna, Mukesh Mohania, and Ekawit Nantajeewarawat, editors, *12th International Conference on Database Systems for Advanced Applications, DASFAA*, volume 1, pages 834–849, Bangkok, Thailand, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-71703-4. doi: 10.1007/978-3-540-71703-4_69.

[131] Jiang Li and Junhu Wang. TwigBuffer : Avoiding Useless Intermediate. In Jayant R Haritsa, Ramamohanarao Kotagiri, and Vikram Pudi, editors, *Database Systems for Advanced Applications: 13th International Conference, DASFAA 2008, New Delhi, India, March 19-21, 2008. Proceedings*, pages 1–8, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78568-2. doi: 10.1007/978-3-540-78568-2_46. URL http://dx.doi.org/10.1007/978-3-540-78568-2{_}46.

[132] Jiang Li and Junhu Wang. Fast Matching of Twig Patterns. In *International Conference on Database and Expert Systems Applications*, volume 5181 LNCS, pages 523–536, 2008. ISBN 3540856536. doi: 10.1007/978-3-540-85654-2_45.

[133] Jiang Li, Junhu Wang, and Maolin Huang. Twig Pattern Matching: A Revisit. In Abdelkader Hameurlain, Stephen W Liddle, Klaus-Dieter Schewe, and Xiaofang Zhou, editors, *Database and Expert Systems Applications: 22nd International Conference, DEXA 2011, Toulouse, France, August 29 - September 2, 2011, Proceedings, Part II*, pages 43–50, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-23091-2. doi: 10.1007/978-3-642-23091-2_4.

[134] Quanzhong Li and B Moon. Indexing and querying XML data for regular path expressions. *Vldb*, pages 361–370, 2001. ISSN 1047-7349. doi: http://www.vldb.org/conf/2001/P361.pdf. URL http://www.vldb.org/conf/2001/P361.pdf.

[135] Wang Lian, David Wai Lok Cheung, Nikos Mamoulis, and Siu Ming Yiu. An Efficient and Scalable Algorithm for Clustering XML Documents by Structure. *IEEE Transactions on Knowledge and Data Engineering*, 16(1):82–96, 2004. ISSN 10414347. doi: 10.1109/TKDE.2004.1264824.

[136] Husheng Liao, Hongyu Gao, and Zhaoning Guan. Recursive Twig Pattern Query. *International Journal of Database Theory and Application*, 7(3):179–190, 2014.

[137] Guiquan Liu, Meiling Yao, Desheng Wang, and Enhong Chen. A novel three-phase XML twig pattern matching algorithm based on version tree. *Proceedings - 2011 8th International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2011*, 3:1678–1688, 2011. doi: 10.1109/FSKD.2011.6019809.

[138] Jian Liu and X. X. Zhang. Dynamic labeling scheme for XML updates. *Knowledge-Based Systems*, 106:135–149, 2016. ISSN 09507051. doi: 10.1016/j.knosys.2016.05.039. URL http://dx.doi.org/10.1016/j.knosys.2016.05.039.

[139] Jian Liu, Z. M. Ma, and Ruizhe Ma. Efficient processing of twig query with compound predicates in fuzzy XML. *Fuzzy Sets and Systems*, 2013. ISSN 01650114. doi: 10.1016/j.fss.2012.11.004.

[140] Jian Liu, Z M Ma, and Li Yan. Efficient labeling scheme for dynamic XML trees. *Information Sciences*, 221(0):338–354, 2013. doi: http://dx.doi.org/10.1016/j.ins.2012.09.036.

[141] Jian Liu, Z. M. Ma, and Xue Feng. Answering ordered tree pattern queries over fuzzy XML data. *Knowledge and Information Systems*, 2014. ISSN 0219-1377. doi: 10.1007/s10115-014-0731-5. URL http://link.springer.com/10.1007/s10115-014-0731-5.

[142] Ziyang Liu, Yichuang Cai, Yi Shan, and Yi Chen. Ranking Friendly Result Composition for XML Keyword Search. In *the 34th International Conference on Conceptual Modeling ER 2015*, volume 9381, pages 441–449, 2015. ISBN 9783319252636. doi: 10.1007/978-3-319-25264-3.

[143] Jiaheng Lu. Towards Benchmarking Multi-Model Databases. In *The 8th biennial Conference on Innovative Data Systems Research (CIDR 2017)*, page 1, Chaminade, California, 2017.

[144] Jiaheng Lu, Ting Chen, and Tok Wang T.W. Ling. Efficient Processing of XML Twig Patterns with Parent Child Edges : A Look-ahead Approach. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 533–542, Washington, D.C., USA, 2004. ACM. ISBN 1581138741. doi: http://doi.acm.org/10.1145/1031171.1031272.

[145] Jiaheng Lu, Tok Wang Ling, Tian Yu, Changqing Li, and Wei Ni. Efficient Processing of Ordered XML Twig Pattern. In Kim Viborg Andersen, John Debenham, and Roland Wagner, editors, *Database and Expert Systems Applications: 16th International Conference, DEXA 2005, Copenhagen, Denmark, August 22-26, 2005. Proceedings*, pages 300–309, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31729-6. doi: 10.1007/11546924_30.

[146] Jiaheng Lu, Tok Wang Ling, Zhifeng Bao, and Chen Wang. Extended XML tree pattern matching: Theories and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 23(3):402–416, 2011. ISSN 10414347. doi: 10.1109/TKDE.2010.126.

[147] Jiaheng Lu, Xiaofeng Meng, and Tok Wang Ling. Indexing and querying XML using extended Dewey labeling scheme. *Data & Knowledge Engineering*, 70(1):35–59, 2011. doi: http://dx.doi.org/10.1016/j.datak.2010.08.001.

[148] Petr Lukas, Radim Bača, and Michal Krátký. Cooking Lightweight XML Query Processor with Binary Joins and Comparing it with Holistic Joins : Technical Report. *the Computing Research Repository (CoRR)*, abs/1703.0, 2017.

[149] Abdul Nizar M and P Sreenivasa Kumar. Efficient Evaluation of Forward XPath Axes over XML Streams. *Society*.

[150] N Mabanza. Analyzing the Impact of XML Storage Models on the Performance of Native XML Database Systems A Case Study. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 210–215, 2010. doi: 10.1109/itng.2010.207.

[151] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*, volume 35. Cambridge University Press, 1 edition, 2008. ISBN 978-0-521-86571-5. doi: 10.1162/coli.2009.35.2.307.

[152] Wim Martens, Frank Neven, Matthias Niewerth, and Thomas Schwentick. BonXai: Combining the Simplicity of DTD with the Expressiveness of XML Schema. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '15, pages 145–156, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2757-2. doi: 10.1145/2745754.2745774.

[153] Mahesh P. Matha. *Core Java: A Comprehensive Study*. Prentice-Hall of India Pvt.Ltd, 1 edition, 2011. ISBN 978-8120342415.

[154] Christian Mathis, Theo Härder, Karsten Schmidt, and Sebastian Bächle. XML indexing and storage: fulfilling the wish list. *Computer Science - Research and Development*, pages 1–18, 2012. doi: 10.1007/s00450-012-0204-6.

[155] Mirjana Mazuran, Elisa Quintarelli, and Letizia Tanca. Data mining for XML query-answering support. *IEEE Transactions on Knowledge and Data Engineering*, 24(8):1393–1407, 2012. ISSN 10414347. doi: 10.1109/TKDE.2011.80.

[156] Catherine C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, New York, NY, USA, 1st edition, 2012. ISBN 0521173019, 9780521173018.

[157] Xiaofeng Meng, Yu Jiang, Yan Chen, and Haixun Wang. XSeq: An Indexing Infrastructure for Tree Pattern Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 941–942, New York, NY, USA, 2004. ACM. ISBN 1-58113-859-8. doi: 10.1145/1007568.1007709. URL http://doi.acm.org/10.1145/1007568.1007709.

[158] Tobias Mettler and Robert Winter. On the Use of Experiments in Design Science Research : A Proposition of an Evaluation Framework. *Communications of the Association for Information Systems*, 34(January):223–240, 2014.

[159] Bertrand Meyer, Christine Choppy, Jørgen Staunstrup, and Jan van Leeuwen. Research evaluation for computer science. *Communications of the ACM*, 52 (4):31, 2009. ISSN 00010782. doi: 10.1145/1498765.1498780. URL http://portal.acm.org/citation.cfm?doid=1498765.1498780.

[160] Michael H. Kay. SAXON The XSLT and XQuery Processor, 2015. URL http://saxon.sourceforge.net/.

[161] Philippe Michiels and George A Mih. Put a Tree Pattern in Your Algebra. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 246–255, 2007. ISBN 1424408032. doi: 10.1109/ICDE.2007.367870.

[162] Gerome Miklau. The University of Washington: XMLData Repository, 2002. URL http://www.cs.washington.edu/research/xmldatasets/.

[163] Salahadin Mohammed, Ahmad F. Barradah, and El Sayed M El-Alfy. Selectivity estimation of extended XML query tree patterns based on prime number labeling and synopsis modeling. *Simulation Modelling Practice and Theory*, 64:30–42, 2015. ISSN 1569190X. doi: 10.1016/j.simpat.2016.01.008.

[164] Mirella M Moro, Zografoula Vagena, and Vassilis J Tsotras. Tree-Pattern Queries on a Lightweight XML Processor. In *Proceedings of the 31st international conference on Very large data bases*, Trondheim, Norway, 2005.

[165] Nadim Nachar. The Mann Whitney U : A Test for Assessing Whether Two Independent Samples Come from the Same Distribution. In *The Quantitative Methods for Psychology*, volume 4, pages 13–20, 2008.

[166] Trygve Nagell. *Introduction to Number Theory*. American Mathematical Society; 2 Reprint edition (June 1, 2001), 2001. ISBN 0828401632.

[167] Richi Nayak and Wina Iryadi. XML schema clustering with semantic and hierarchical similarity measures. *Knowledge-Based Systems*, 20(4):336–349, 2007. doi: http://dx.doi.org/10.1016/j.knosys.2006.08.006.

[168] Robert G. Newcombe. Confidence intervals for an effect size measure based on the Mann-Whitney statistic. Part 1: General issues and tail-area-based methods. *Statistics in Medicine*, 25(4):543–557, 2006. ISSN 02776715. doi: 10.1002/sim.2323.

[169] M Nicola and J John. XML Parsing: A Threat to Database Performance. In *the twelfth international conference on ACM Digitail Library*, volume 9, pages 3–6, 2003. ISBN 1581137230. doi: 10.1145/956863.956898.

[170] Matthias Nicola, Irina Kogan, and Berni Schiefer. An XML transaction processing benchmark. *Proceedings of the 2007 ACM SIGMOD international conference on Management of data - SIGMOD '07*, page 937, 2007. ISSN 07308078. doi: 10.1145/1247480.1247590. URL http://portal.acm.org/citation.cfm?doid=1247480.1247590.

[171] Bo Ning and Chengfei Liu. XML filtering with XPath expressions containing parent and ancestor axes. *Information Sciences*, 210(0):41–54, 2012. doi: http://dx.doi.org/10.1016/j.ins.2012.04.035.

[172] Bo Ning, Guoren Wang, and Jeffrey Xu Yu. A Holistic Algorithm for Efficiently Evaluating Xtwig Joins. In *Database Systems for Advanced Applications: 13th International Conference, DASFAA*, volume 1, pages 571–572, New Delhi, India, 2008.

[173] M Abdul Nizar and P Sreenivasa Kumar. Order-Aware Twigs : Adding Order Semantics to Twigs. *Journal of Information and Data Management*, 3(1):3–17, 2012.

[174] S. Noor Ea Thahasin and P. Jayanthi. Vector based labeling method for dynamic XML documents. In *2013 International Conference on Information Communication and Embedded Systems, ICICES 2013*, pages 217–221, 2013. ISBN 9781467357869. doi: 10.1109/ICICES.2013.6508390.

[175] Martin F O'Connor and Mark Roantree. SCOOTER: A Compact and Scalable Dynamic Labeling Scheme for XML Updates. In Stephen W Liddle, Klaus-Dieter Schewe, A Min Tjoa, and Xiaofang Zhou, editors, *Database and Expert Systems Applications. DEXA 2012. Lecture Notes in Computer Science*, pages 26–40. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32600-4. doi: 10.1007/978-3-642-32600-4_4.

[176] Peter Ogden, David Thomas, and Peter Pietzuch. Scalable XML Query Processing using Parallel Pushdown Transducers. *the VLDB Endowment*, 6(14):1738–1749, 2013.

[177] Eric Pardede, J. Wenny Rahayu, and David Taniar. XML-enabled relational database for XML document update. *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*, 2:205–209, 2006. ISSN 1550445X. doi: 10.1109/AINA.2006.354.

[178] Richard Pattis. Chapter 1 EBNF : A Notation to Describe Syntax. In *the Ada Programming Language*, pages 1–19. Dreamsongs Press, 2013.

[179] Ken Peffers, Tuure Tuunanen, Marcus Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, 2007. ISSN 0742-1222. doi: 10.2753/MIS0742-1222240302.

[180] Jaroslav Pokorny. XML A Challenge for Databases? *Contemporary Trends in Systems Development*, pages 147–148, 2001.

[181] Chung Keung Poon and Leo Yuen. Faster Twig Pattern Matching Using Extended Dewey ID. In Stéphane Bressan, Josef Küng, and Roland Wagner, editors, *Database and Expert Systems Applications: 17th International Conference, DEXA 2006, Kraków, Poland, September 4-8, 2006.*, pages 297–306, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-37872-3. doi: 10.1007/11827405_29.

[182] GNU Project. The R Project for Statistical Computing, 2017. URL https://www.r-project.org/.

[183] The Apache Xerces™ Project. Xerces-C++ XML Parser, 1999. URL https://xerces.apache.org/xerces-c/.

[184] Ghassan Z. Qadah. Indexing techniques for processing generalized XML documents. *Computer Standards and Interfaces*, 49:34–43, 2017. ISSN 09205489. doi: 10.1016/j.csi.2016.07.002.

[185] Lu Qin, Jeffrey Xu Yu, and Bolin Ding. TwigList: Make Twig Pattern Matching Fast. In Ramamohanarao Kotagiri, P Radha Krishna, Mukesh Mohania, and Ekawit Nantajeewarawat, editors, *12th International Conference on Database Systems for Advanced Applications, DASFAA*, pages 850–862, Bangkok, Thailand, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-71703-4. doi: 10.1007/978-3-540-71703-4_70.

[186] Zunyue Qin, Yong Tang, Feiyi Tang, Jing Xiao, Changqin Huang, and Hongzhi Xu. Efficient XML Query and Update Processing Using A Novel Prime-Based Middle Fraction Labeling Scheme. *China Communications*, 14(March):145–157, 2017.

[187] Amjad Qtaish and Kamsuriah Ahmad. XAncestor: An efficient mapping approach for storing and querying XML documents in relational database using path-based technique. *Knowledge-Based Systems*, 114(October):167–192, 2016. ISSN 09507051. doi: 10.1016/j.knosys.2016.10.009.

[188] Praveen Rao and B. Moon. PRIX: indexing and querying XML using prufer sequences. In *the 20th International Conference on Data Engineering*, number January, pages 288–299, 2004. ISBN 1063-6382. doi: 10.1109/icde.2004.1320005.

[189] Praveen Rao and Bongki Moon. Sequencing XML Data and Query Twigs for Fast Pattern Matching. *ACM Trans. Database Syst.*, 31(1):299–345, mar 2006. ISSN 0362-5915. doi: 10.1145/1132863.1132871. URL http://doi.acm.org/10.1145/1132863.1132871.

[190] Chen Rongxin and Chen Weibin. A parallel solution to XML query application. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 6, pages 542–546, 2010. doi: 10.1109/iccsit.2010.5564719.

[191] Kanda Runapongsa and Jignesh Patel. Storing and Querying XML Data in Object-Relational DBMSs. *XMLBased Data Management and Multimedia Engineering EDBT 2002 Workshops*, pages 579–583, 2002. ISSN 0302-9743. URL http://dx.doi.org/10.1007/3-540-36128-6{_}15.

[192] Kanda Runapongsa, Jignesh M. Patel, H. V. Jagadish, Yun Chen, and Shurug Al-Khalifa. The Michigan benchmark: Towards XML query performance diagnostics. *Information Systems*, 31(2):73–97, 2006. ISSN 03064379. doi: 10.1016/j.is.2004.09.004.

[193] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009. ISSN 13823256. doi: 10.1007/s10664-008-9102-8.

[194] SAX: the Simple A P I for X M L SAX Project Organization. SAX, 2014. URL http://www.saxproject.org/.

[195] Albrecht Schmidt, Florian Waas, Martin Kersten, Ralph Busse, Michael J Carey, and G B Amsterdam. XMark : A Benchmark for XML Data Management. In *VLDB '02 Proceedings of the 28th international conference on Very Large Data Bases*, pages 974–985, 2002.

[196] Karsten Schmidt, Sebastian Bächle, and Theo Härder. Benchmarking performance-critical components in a native XML database system. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5667 LNCS:64–78, 2009. ISSN 03029743. doi: 10.1007/978-3-642-04205-8_7.

[197] Kai Schweinsberg and Lutz Wegner. Advantages of complex SQL types in storing XML documents. *Future Generation Computer Systems*, 68:500–507, 2017. ISSN 0167739X. doi: 10.1016/j.future.2016.02.013.

[198] Mirit Shalem and Ziv Bar-Yossef. The space complexity of processing XML twig queries over indexed documents. In *Proceedings - International Conference on Data Engineering*, number March, pages 824–832, 2008. ISBN 9781424418374. doi: 10.1109/ICDE.2008.4497491.

[199] M A Shao-long, Wang Xin-jun, and Zhang Feng. Efficient Processing of XML Twig Pattern Matching based on Extended Region Encoding Labeling Scheme. In *IEEE International Symposium on IT in Medicine & Education*, pages 1–6, 2009. ISBN 9781424439300.

[200] Mary Shaw. What makes good research in software engineering? *International Journal on Software Tools for Technology ...*, 4(1):1–7, 2002. ISSN 1433-2779. doi: 10.1007/s10009-002-0083-4. URL http://link.springer.com/article/10.1007/s10009-002-0083-4.

[201] Mikael Fernandus Simalango. XML Query Processing and Query Languges: A Survey. 2010.

[202] David S.Johnson. A Theoretician's Guide to Experimental Analysis of Algorithms. In *Proceedings of the 5th and 6th DIMACS Implementation Challenges*, pages 215–250, 2002.

[203] Aaron Skonnard and Martin Gudgin. *Essential Xml Quick Reference: A Programmer's Reference to XML, XPath, XSLT, XML Schema, SOAP, and More*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2001. ISBN 0201740958.

[204] Samini Subramaniam, Su-Cheng Cheng Haw, and Poo Kuan Hoong. S-XML: An efficient mapping scheme to bridge XML and relational database. *Knowledge-Based Systems*, 27(0):369–380, 2012. ISSN 09507051. doi: 10.1016/j.knosys.2011.11.007.

[205] Samini Subramaniam, Su-Cheng Haw, and Lay-Ki Soon. DGReLab + : Improving XML Path Query Processing by Avoiding Buffering Irrelevant Results. In *7th International Conference on Advances in Computing & Communications, ICACC-2017*, volume 115, pages 804–811. Elsevier B.V., 2017. doi: 10.1016/j.procs.2017.09.157.

[206] Samini Subramaniam, Su-cheng Haw, Lay-ki Soon, and Kok-leong Koong. QTwig: A Structural Join Algorithm for Efficient Query Retrieval Based on Region-Based Labeling. *International Journal of Software Engineering and Knowledge Engineering*, 27(2):321–342, 2017.

[207] G V Subramanyam, P Sreenivasa Kumar, Jayant Haritsa, and T M Vijayaraman Editors. Efficient Handling of Sibling Axis in XPath. In *Advances in Data Management 2005, Proceedings of the Eleventh International Conference on Management of Data, January 6, 7, and 8, 2005, Goa, India*, pages 95–102, 2005.

[208] Suphat Sukamolson. Fundamentals of quantitative research. *Language Institute*, page 20, 2007. doi: 9781848608641.

[209] Mohammed Amin Tahraoui, Karen Pinel-Sauvagnat, Cyril Laitang, Mohand Boughanem, Hamamache Kheddouci, and Lei Ning. A survey on tree matching and XML retrieval. *Computer Science Review*, 8(0):1–23, 2013. doi: http://dx.doi.org/10.1016/j.cosrev.2013.02.001.

[210] Hideaki Takeda, Hideaki Takeda, Paul Veerkamp, Paul Veerkamp, Tetsuo Tomiyama, Tetsuo Tomiyama, Hiroyuki Yoshikawa, and Hiroyuki Yoshikawa. Modeling Design Processes. *AI Magazine*, 11(4):37–48, 1990. ISSN 0738-4602. doi: 10.1609/aimag.v11i4.855.

[211] Igor Tatarinov, Stratis D Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system, 2002.

[212] Shirish Tatikonda and Srinivasan Parthasarathy. Hashing tree-structured data: Methods and applications. *Proceedings - International Conference on Data Engineering*, pages 429–440, 2010. ISSN 10844627. doi: 10.1109/ICDE.2010.5447882.

[213] Shirish Tatikonda, Shirish Tatikonda, Srinivasan Parthasarathy, Srinivasan Parthasarathy, Matthew Goyder, and Matthew Goyder. LCS-TRIM: dynamic programming meets XML indexing and querying. In *the 33rd international conference on Very large data bases VLDB*, pages 63–74, 2007. ISBN 978-1-59593-649-3.

[214] Joe Tekli, Richard Chbeir, and Kokou Yetongnon. Efficient XML Structural Similarity Detection using Sub-tree Commonalities. In *22nd Brazilian symposium on databases (SBBD)*, pages 116–130, 2007.

[215] Terrill Brett Spell. *Pro Java 8 Programming*. Apress, 1st edition, 2015. ISBN 978-1484206423.

[216] David R. Tobergte and Shirley Curtis. Experimentation in Software Engineering Book. *Journal of Chemical Information and Modeling*, 53(9):1689–1699, 2013. ISSN 1098-6596. doi: 10.1017/CBO9781107415324.004.

[217] Zografoula Vagena, Nick Koudas, Divesh Srivastava, and V. Tsotras. Efficient handling of positional predicates within XML query processing. *Proc. 3rd Int. XML Database Symp. on Database and XML Technologies*, pages 579–579, 2005. ISSN 03029743. URL http://www.springerlink.com/index/82augt6wrxff4p8q.pdf.

[218] Zografoula Vagena, Nick Koudas, Divesh Srivastava, and Vassilis J Tsotras. Answering Order-based Queries over XML Data. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, number January in WWW '05, pages 1162–1163, New York, NY, USA, 2005. ACM. ISBN 1-59593-051-5. doi: 10.1145/1062745.1062919. URL http://doi.acm.org/10.1145/1062745.1062919.

[219] Andris Vargha and Harold D Delaney. The Kruskal-Wallis Test and Stochastic Homogeneity. *Journal of Educational and Behavioral Statistics*, 23(2):170–192, 1998.

[220] W3C. XSL Transformations (XSLT) Version 1.0, 1999. URL https://www.w3.org/TR/xslt.

[221] W3C. XML Pointer Language (XPointer) Version 1.0, 2001. URL https://www.w3.org/TR/WD-xptr.

[222] W3C. XML Path Language (XPath) 2.0 (Second Edition), 2010. URL http://www.w3.org/TR/xpath20/.

[223] W3C. SCHEMA, 2012. URL http://www.w3.org/standards/xml/schema.

[224] W3C. XQuery 3.0: An XML Query Language, 2014. URL http://www.w3.org/TR/xquery-30/.

[225] W3C. Extensible Markup Language (XML), 2016. URL http://www.w3.org/XML/.

[226] W3schools. Introduction to XML. URL http://www.w3schools.com/xml/xml{_}whatis.asp.

[227] W3schools. XQuery Tutorial, 2016. URL http://www.w3schools.com/xquery/.

[228] Dale Waldt. Six strategies for extending XML schemas in a single namespace Create flexible XML schemas that grow to fit changing. Technical report, 2010.

[229] Bing Wang, Xianfeng Liu, Zhenxi Lei, and Gaocai Wang. CCSU: Research on the Compression Coding of Supporting Data Update Completely. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 1796–1801, 2013. doi: 10.1109/HPCC.and.EUC.2013.257.

[230] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 110–121, New York, NY, USA, 2003. ACM. ISBN 1-58113-634-X. doi: 10.1145/872757.872774. URL http://doi.acm.org/10.1145/872757.872774.

[231] Lu Wei, K Chiu, and Pan Yinfei. A Parallel Approach to XML Parsing. In *Grid Computing, 7th IEEE/ACM International Conference on*, pages 223–230, 2006. doi: 10.1109/icgrid.2006.311019.

[232] Zhang Wei and R van Engelen. A Table-Driven Streaming XML Parsing Methodology for High-Performance Web Services. In *Web Services, 2006. ICWS '06. International Conference on*, pages 197–204, 2006. doi: 10.1109/icws.2006.15.

[233] Adrian White. HOW TO GET A PhD: A HANDBOOK FOR STUDENTS AND THEIR SUPERVISORS. *Complementary Therapies in Medicine*, 11(1):51, 2003. ISSN 09652299. doi: 10.1016/S0965-2299(03)00003-7.

[234] Carrie Williams. Research Methods. *Journal of Business & Economic Research*, 5 (3):65–72, 2007. ISSN 0895-4356. doi: 10.1093/fampract/cmi221.

[235] Huayu Wu, Tok Wang Ling, and Gillian Dobbie. TP + Output : Modeling Complex Output Information in XML Twig Pattern Query. In *the 7th International XML Database Symposium, XSym*, pages 128–143, Singapore, 2010.

[236] Huayu Wu, Tok Wang Ling, Bo Chen, and Liang Xu. TwigTable : using semantics in XML twig pattern query processing. *Journal on Data Semantics XV*, 6720: 102–129, 2011.

[237] Huayu Wu, Chunbin Lin, Tok Wang Ling, and Jiaheng Lu. Processing XML twig pattern query with wildcards. In Stephen W Liddle, Klaus-Dieter Schewe, A Min Tjoa, and Xiaofang Zhou, editors, *International Conference on Database and Expert Systems Applications*, volume 7446 LNCS, pages 326–341, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 9783642325991. doi: 10.1007/978-3-642-32600-4_24.

[238] X Wu, Mong Li Lee, and W Hsu. A prime number labeling scheme for dynamic ordered XML trees. In *Proceedings. 20th International Conference on Data Engineering*, pages 66–78, 2004. ISBN 1063-6382 VO -. doi: 10.1109/ICDE.2004.1319985.

[239] Xiaoying Wu and Dimitri Theodoratos. A survey on XML streaming evaluation techniques. *The VLDB Journal*, 22(2):177–202, 2013. doi: 10.1007/s00778-012-0281-y.

[240] Xin Wu and Guiquan Liu. XML twig pattern matching using version tree. *Data and Knowledge Engineering*, 64(3):580–599, 2008. doi: http://dx.doi.org/10.1016/j.datak.2007.09.013.

[241] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *International Conference on Data Engineering*, pages 443–454, 2003. ISBN 0-7803-7665-X. doi: 10.1109/ICDE.2003.1260812.

[242] Wu Xiaoying, S Souldatos, D Theodoratos, T Dalamagas, Y Vassiliou, and T Sellis. Processing and Evaluating Partial Tree Pattern Queries on XML Data. *Knowledge and Data Engineering, IEEE Transactions on*, 24(12):2244–2259, 2012. doi: 10.1109/tkde.2011.137.

[243] Liang Xu, Tok Wang Ling, Huayu Wu, and Zhifeng Bao. DDE: from dewey to a fully dynamic XML labeling scheme. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 719–730, 2009. doi: 10.1145/1559845.1559921.

[244] Liang Xu, Tok Wang Ling, Huayu Wu, Xu Liang, Ling Tok Wang, and Wu Huayu. Labeling Dynamic XML Documents: An Order-Centric Approach. In *IEEE Transactions on Knowledge and Data Engineering*, volume 24, pages 100–113, 2012. ISBN 1041-4347. doi: 10.1109/tkde.2010.221.

[245] Xiaoshuang Xu, Yucai Feng, and Feng Wang. Efficient processing of XML twig queries with all predicates. In *Proceedings of the 2009 8th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2009*, 2009. ISBN 9780769536415. doi: 10.1109/ICIS.2009.74.

[246] Beverly Yang, Marcus Fontoura, Eugene Shekita, Sridhar Rajagopalan, and Kevin Beyer. Virtual Cursors for XML Joins. In *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management*, CIKM '04, pages 523–532, New York, NY, USA, 2004. ACM. ISBN 1-58113-874-1. doi: 10.1145/1031171.1031271. URL http://doi.acm.org/10.1145/1031171.1031271.

[247] Benjamin Bin Yao, M. Tamer Özsu, and Nitin Khandelwal. XBench benchmark and performance testing of XML DBMSs. *Proceedings - International Conference on Data Engineering*, 20:621–632, 2004. ISSN 1063-6382. doi: 10.1109/ICDE.2004.1320032.

[248] Xiang Yongqing, Deng Zhihong, Yu Hang, Wang Sijing, and Gao Ning. A new indexing strategy for XML keyword search. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2010 Seventh International Conference on*, volume 5, pages 2412–2416, 2010. doi: 10.1109/fskd.2010.5569522.

[249] Tian Yu, TokWang Wang Ling, and Jiaheng Lu. TwigStackList ¬: A Holistic Twig Join Algorithm for Twig Query with Not-Predicates on XML Data. In Mong Li Lee, Kian-Lee Tan, and Vilas Wuwongse, editors, *Database Systems for Advanced Applications*, volume 3882, pages 249–263. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-33337-1. doi: 10.1007/11733836_19.

[250] Leo Yuen and Chung Keung Poon. Relational Index Support for XPath Axes. *Database and XML Technologies, Third International XML Database Symposium, XSym 2005, Trondheim, Norway, August 28-29, 2005, Proceedings. Lecture Notes in Computer Science*, 5679(August 2005), 2009. doi: 10.1007/978-3-642-03555-5.

[251] Hanaa Al Zadjali and Siobhán North. XML Labels Compression using Prefix-Encodings. In *The 12th International Conference on Web Information Systems and Technologies (WEBIST 2016)*, pages 69–75, Rome, Italy, 2016. doi: 10.5220/0005755500690075.

[252] Qiang Zeng and Hai Zhuge. Stack-based Algorithms for Pattern Matching on DAGs. In *the 31st VLDB Conference*, Trondheim, Norway, 2005. ISBN 1595931546.

[253] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. *ACM SIGMOD Record*, 30:425–436, 2001. ISSN 01635808. doi: 10.1145/376284.375722.

[254] Chen Zhiyuan, J Gehrke, F Korn, N Koudas, J Shanmugasundaram, and D Srivastava. Index Structures for Matching XML Twigs Using Relational Query Processors. In *Data Engineering Workshops, 2005. 21st International Conference on*, page 1273, 2005. doi: 10.1109/icde.2005.231.

[255] Junfeng Zhou, Min Xie, and Xiaofeng Meng. TwigStack+: Holistic twig join pruning using extended solution extension. *Wuhan University Journal of Natural Sciences*, 12(5):855–860, 2007. ISSN 10071202. doi: 10.1007/s11859-007-0032-x.

Appendix A

Top-Down Holistic Approaches Full results

Table A.1: Results for paired comparisons based on the U test over the DBLP dataset.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	Median A	Median B	effect size r	Best
TwigStackList	TwigStackPrime	DQ1	2.26E-34	TRUE	0.34	0.237	0.8605028	TwigStackPrime
TwigStackList	TwigStackPrime	DQ2	4.33E-18	TRUE	1.035	1.009	0.607437	TwigStackPrime
TwigStackList	TwigStackPrime	DQ3	4.55E-30	TRUE	0.21	0.196	0.8012964	TwigStackPrime
TwigStackList	TwigStackPrime	DQ4	6.68E-32	TRUE	0.209	0.133	0.8270301	TwigStackPrime
TwigStackList	TwigStack	DQ1	4.46E-33	TRUE	0.34	0.299	0.8431224	TwigStack
TwigStackList	TwigStack	DQ2	1.52E-08	TRUE	1.035	1.022	0.3916347	TwigStack
TwigStackList	TwigStack	DQ3	5.96E-31	TRUE	0.21	0.195	0.8137879	TwigStack
TwigStackList	TwigStack	DQ4	4.51E-15	TRUE	0.209	0.1975	0.5481811	TwigStack
TwigStackPrime	TwigStack	DQ1	3.63E-33	TRUE	0.237	0.299	0.8443291	TwigStackPrime
TwigStackPrime	TwigStack	DQ2	2.00E-09	TRUE	1.009	1.022	0.416049	TwigStackPrime
TwigStackPrime	TwigStack	DQ3	0.005262825	TRUE	0.196	0.195	0.1808825	TwigStack
TwigStackPrime	TwigStack	DQ4	2.35E-31	TRUE	0.133	0.1975	0.8194512	TwigStackPrime

Table A.2: Results for paired comparisons based on the U test over the XMark dataset.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	Median A	Median B	effect size r	Best
TwigStackList	TwigStackPrime	XQ1	2.12E-33	TRUE	14.3205	14.039	0.8474823	TwigStackPrime
TwigStackList	TwigStackPrime	XQ2	0.2082787	FALSE	0.216	0.216	0.05744595	tie
TwigStackList	TwigStackPrime	XQ3	0.676971	FALSE	52.3595	52.5035	0.03247355	tie
TwigStackList	TwigStackPrime	XQ4	9.54E-31	TRUE	21.5035	20.975	0.810919	TwigStackPrime
TwigStackList	TwigStackPrime	XQ5	6.85E-06	TRUE	58.618	58.4545	0.3074949	TwigStackPrime
TwigStackList	TwigStackPrime	XQ6	0.8306972	FALSE	8.308	8.3165	0.06766476	tie
TwigStackList	TwigStack	XQ1	2.56E-34	TRUE	14.3205	23.939	0.859799	TwigStackList
TwigStackList	TwigStack	XQ2	1.47E-09	TRUE	0.216	0.215	0.419672	TwigStack
TwigStackList	TwigStack	XQ3	2.62E-18	TRUE	52.3595	53.9075	0.6114935	TwigStackList
TwigStackList	TwigStack	XQ4	5.66E-32	TRUE	21.5035	20.953	0.8280259	TwigStack
TwigStackList	TwigStack	XQ5	1.17E-31	TRUE	58.618	59.9775	0.8236795	TwigStackList
TwigStackList	TwigStack	XQ6	1.03E-11	TRUE	8.308	8.2205	0.4739111	TwigStack
TwigStackPrime	TwigStack	XQ1	2.56E-34	TRUE	14.039	23.939	0.8597981	TwigStackPrime
TwigStackPrime	TwigStack	XQ2	2.17E-06	TRUE	0.216	0.215	0.3248566	TwigStack
TwigStackPrime	TwigStack	XQ3	3.21E-19	TRUE	52.5035	53.9075	0.6282234	TwigStackPrime
TwigStackPrime	TwigStack	XQ4	0.2518079	FALSE	20.975	20.953	0.0472921	tie
TwigStackPrime	TwigStack	XQ5	3.72E-32	TRUE	58.4545	59.9775	0.8305387	TwigStackPrime
TwigStackPrime	TwigStack	XQ6	4.80E-12	TRUE	8.3165	8.2205	0.4817012	TwigStack

Table A.3: Results for paired comparisons based on the U test over the TreeBank dataset.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	Median A	Median B	effect size r	Best
TwigStackList	TwigStackPrime	TQ1	1.51E-31	TRUE	0.139	0.127	0.8221121	TwigStackPrime
TwigStackList	TwigStackPrime	TQ2	6.70E-34	TRUE	1.52	1.466	0.8542129	TwigStackPrime
TwigStackList	TwigStackPrime	TQ3	9.22E-35	TRUE	0.409	0.462	0.8656703	TwigStackList
TwigStackList	TwigStackPrime	TQ4	3.56E-24	TRUE	1.623	1.61	0.7124215	TwigStackPrime
TwigStackList	TwigStackPrime	TQ5	2.55E-34	TRUE	5.985	8.2285	0.859816	TwigStackList
TwigStackList	TwigStackPrime	TQ6	1.06E-25	TRUE	36.53	35.9195	0.7364704	TwigStackPrime
TwigStackList	TwigStackPrime	TQ7	2.42E-34	TRUE	1.568	1.494	0.860124	TwigStackPrime
TwigStackList	TwigStackPrime	TQ8	2.49E-34	TRUE	3.274	3.017	0.859955	TwigStackPrime
TwigStackList	TwigStackPrime	TQ9	2.46E-34	TRUE	2.189	2.324	0.8600162	TwigStackList
TwigStackList	TwigStackPrime	TQ10	3.52E-45	TRUE	1107.155	850.555	0.9939336	TwigStackPrime
TwigStackList	TwigStack	TQ1	1.08E-33	TRUE	0.139	0.126	0.8514198	TwigStack
TwigStackList	TwigStack	TQ2	2.32E-34	TRUE	1.52	1.785	0.860359	TwigStackList
TwigStackList	TwigStack	TQ3	8.98E-35	TRUE	0.409	0.434	0.8658218	TwigStackList
TwigStackList	TwigStack	TQ4	2.19E-34	TRUE	1.623	2.785	0.8606963	TwigStackList
TwigStackList	TwigStack	TQ5	2.55E-34	TRUE	5.985	10.4385	0.8598241	TwigStackList
TwigStackList	TwigStack	TQ6	5.64E-39	TRUE	36.53	1371.105	0.9196807	TwigStackList
TwigStackList	TwigStack	TQ7	2.45E-34	TRUE	1.568	1.969	0.8600536	TwigStackList
TwigStackList	TwigStack	TQ8	2.48E-34	TRUE	3.274	2.7955	0.8599869	TwigStack
TwigStackList	TwigStack	TQ9	2.45E-34	TRUE	2.189	2.056	0.8600553	TwigStack
TwigStackList	TwigStack	TQ10	3.52E-45	TRUE	1107.155	1151.6	0.9939336	TwigStackList
TwigStackPrime	TwigStack	TQ1	9.59E-18	TRUE	0.127	0.126	0.6009459	TwigStack
TwigStackPrime	TwigStack	TQ2	2.34E-34	TRUE	1.466	1.785	0.8603133	TwigStackPrime
TwigStackPrime	TwigStack	TQ3	1.04E-34	TRUE	0.462	0.434	0.8649838	TwigStack
TwigStackPrime	TwigStack	TQ4	2.13E-34	TRUE	1.61	2.785	0.8608599	TwigStackPrime
TwigStackPrime	TwigStack	TQ5	2.55E-34	TRUE	8.2285	10.4385	0.859816	TwigStackPrime
TwigStackPrime	TwigStack	TQ6	5.63E-39	TRUE	35.9195	1371.105	0.9196842	TwigStackPrime
TwigStackPrime	TwigStack	TQ7	2.43E-34	TRUE	1.494	1.969	0.8600891	TwigStackPrime
TwigStackPrime	TwigStack	TQ8	2.47E-34	TRUE	3.017	2.7955	0.8600032	TwigStack
TwigStackPrime	TwigStack	TQ9	2.43E-34	TRUE	2.324	2.056	0.8600963	TwigStack
TwigStackPrime	TwigStack	TQ10	3.52E-45	TRUE	850.555	1151.6	0.9939336	TwigStackPrime

Table A.4: Results for paired comparisons based on the U test over the Random dataset.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	Median A	Median B	Effect size r	Best
TwigStackList	TwigStackPrime	RQ1	2.55E-34	TRUE	6.9775	7.618	0.8598065	TwigStackList
TwigStackList	TwigStackPrime	RQ2	2.56E-34	TRUE	94.542	87.5205	0.859789	TwigStackPrime
TwigStackList	TwigStackPrime	RQ3	2.56E-34	TRUE	44.219	41.8955	0.8597903	TwigStackPrime
TwigStackList	TwigStackPrime	RQ4	0.06142632	FALSE	409.3945	412.4995	0.1091002	tie
TwigStackList	TwigStackPrime	RQ5	5.57E-34	TRUE	12.318	11.9135	0.8552907	TwigStackPrime
TwigStackList	TwigStackPrime	RQ6	3.89E-23	TRUE	1216.978	769.7405	0.6955955	TwigStackPrime
TwigStackList	TwigStackPrime	RQ7	9.26E-26	TRUE	7.8395	8.3285	0.7373509	TwigStackList
TwigStackList	TwigStackPrime	RQ8	2.55E-34	TRUE	11.7885	12.2725	0.8598202	TwigStackList
TwigStackList	TwigStackPrime	RQ9	2.56E-34	TRUE	43.0745	23.394	0.8597942	TwigStackPrime
TwigStackList	TwigStack	RQ1	2.55E-34	TRUE	6.9775	6.6865	0.8598095	TwigStack
TwigStackList	TwigStack	RQ2	2.56E-34	TRUE	94.542	131.433	0.859788	TwigStackList
TwigStackList	TwigStack	RQ3	2.56E-34	TRUE	44.219	69.612	0.8597903	TwigStackList
TwigStackList	TwigStack	RQ4	2.56E-34	TRUE	409.3945	509.6845	0.859787	TwigStackList
TwigStackList	TwigStack	RQ5	2.55E-34	TRUE	12.318	11.223	0.8598075	TwigStack
TwigStackList	TwigStack	RQ6	5.02E-33	TRUE	1216.978	4032.539	0.8424288	TwigStackList
TwigStackList	TwigStack	RQ7	2.56E-34	TRUE	7.8395	12.503	0.8598	TwigStackList
TwigStackList	TwigStack	RQ8	2.54E-34	TRUE	11.7885	8.9575	0.8598296	TwigStack
TwigStackList	TwigStack	RQ9	2.56E-34	TRUE	43.0745	510.318	0.8597886	TwigStackList
TwigStackPrime	TwigStack	RQ1	2.55E-34	TRUE	7.618	6.6865	0.8598069	TwigStack
TwigStackPrime	TwigStack	RQ2	2.56E-34	TRUE	87.5205	131.433	0.8597886	TwigStackPrime
TwigStackPrime	TwigStack	RQ3	2.56E-34	TRUE	41.8955	69.612	0.8597896	TwigStackPrime
TwigStackPrime	TwigStack	RQ4	2.56E-34	TRUE	412.4995	509.6845	0.859787	TwigStackPrime
TwigStackPrime	TwigStack	RQ5	2.56E-34	TRUE	11.9135	11.223	0.8598007	TwigStack
TwigStackPrime	TwigStack	RQ6	2.56E-34	TRUE	769.7405	4032.539	0.859787	TwigStackPrime
TwigStackPrime	TwigStack	RQ7	2.56E-34	TRUE	8.3285	12.503	0.859799	TwigStackPrime
TwigStackPrime	TwigStack	RQ8	2.54E-34	TRUE	12.2725	8.9575	0.8598303	TwigStack
TwigStackPrime	TwigStack	RQ9	2.56E-34	TRUE	23.394	510.318	0.8597925	TwigStackPrime

Appendix B

Ordered Top-Down Holistic Approaches Full results

Tables B.1, B.2 and B.3 present the XPath expressions used in the experiments of Chapter 7.

Table B.1: Experimental ordered TPQs for XMark.

Code	XPath expression
OXQ_1	//mail/text/keyword/following-sibling::bold
OXQ_2	//text/bold[//keyword]/following::emph
OXQ_3	//description//text/following::parlist
OXQ_4	//listitem//bold/following-sibling::text[//emph]//keyword
OXQ_5	//mail/text[/keyword]//bold[keyword«bold]

Table B.2: Experimental ordered TPQs for TreeBank.

Code	XPath expression
OTQ_1	//S/VP/PP/IN/following::NP/following::VBN
OTQ_2	//NP/NN/following-sibling::PP
OTQ_3	//VP/DT/following::PRP_DOLLAR_
OTQ_4	//NP/NN/preceding-sibling::PP
OTQ_5	//PP//NP[/preceding::VP]/preceding-sibling::VBN
OTQ_6	//PP//NP[/preceding::VP]/VBN
OTQ_7	//PP//NP[/preceding::VP]/preceding::VBN
OTQ_8	//VP/NP//NNS/following::S
OTQ_9	//S[//MD]//ADJ[MD«ADJ]

Table B.4: Results for paired comparisons based on the U test over the XMark dataset.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	Median A	Median B	effect size	Winner
SFTwigStack	SFTwigStackList	OXQ1	2.56E-34	TRUE	42.869	35.3105	0.8597906	SFTwigStackList
SFTwigStack	SFTwigStackList	OXQ2	0.2720523	FALSE	8.5085	8.5255	0.04289436	tie

Table B.4: Results for paired comparisons based on the U test over the XMark dataset.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	Median A	Median B	effect size	Winner
SFTwigStack	SFTwigStackList	OXQ3	3.52E-45	TRUE	622.7475	639.935	0.9939336	SFTwigStack
SFTwigStack	SFTwigStackList	OXQ4	3.52E-45	TRUE	510.586	516.767	0.9939336	SFTwigStack
SFTwigStack	SFTwigStackList	OXQ5	3.55E-26	TRUE	49.549	45.47	0.7437648	SFTwigStackList
SFTwigStack	SFTwigStackPrime	OXQ1	2.56E-34	TRUE	42.869	35.1695	0.8597906	SFTwigStackPrime
SFTwigStack	SFTwigStackPrime	OXQ2	2.54E-32	TRUE	8.5085	8.805	0.8328053	SFTwigStack
SFTwigStack	SFTwigStackPrime	OXQ3	3.52E-45	TRUE	622.7475	628.899	0.9939336	SFTwigStack
SFTwigStack	SFTwigStackPrime	OXQ4	3.52E-45	TRUE	510.586	517.802	0.9939336	SFTwigStack
SFTwigStack	SFTwigStackPrime	OXQ5	3.99E-23	TRUE	49.549	45.5235	0.6954218	SFTwigStackPrime
SFTwigStack	OTJPrime	OXQ1	2.56E-34	TRUE	42.869	18.1765	0.8597951	OTJPrime
SFTwigStack	OTJPrime	OXQ2	2.55E-34	TRUE	8.5085	3.9685	0.8598052	OTJPrime
SFTwigStack	OTJPrime	OXQ3	5.64E-39	TRUE	622.7475	77.632	0.9196779	OTJPrime
SFTwigStack	OTJPrime	OXQ4	5.64E-39	TRUE	510.586	17.0415	0.9196787	OTJPrime
SFTwigStack	OTJPrime	OXQ5	2.56E-34	TRUE	49.549	24.612	0.8597877	OTJPrime
SFTwigStack	OTJPrimeList	OXQ1	2.56E-34	TRUE	42.869	14.0375	0.8597948	OTJPrimeList
SFTwigStack	OTJPrimeList	OXQ2	2.56E-34	TRUE	8.5085	2.891	0.8598026	OTJPrimeList
SFTwigStack	OTJPrimeList	OXQ3	5.64E-39	TRUE	622.7475	31.3805	0.9196791	OTJPrimeList
SFTwigStack	OTJPrimeList	OXQ4	5.60E-39	TRUE	510.586	1.003	0.9197196	OTJPrimeList
SFTwigStack	OTJPrimeList	OXQ5	2.56E-34	TRUE	49.549	21.9755	0.8597877	OTJPrimeList
SFTwigStack	OTJPMultiLists	OXQ1	2.56E-34	TRUE	42.869	14.204	0.8597948	OTJPMultiLists
SFTwigStack	OTJPMultiLists	OXQ2	2.55E-34	TRUE	8.5085	2.996	0.8598153	OTJPMultiLists
SFTwigStack	OTJPMultiLists	OXQ3	5.64E-39	TRUE	622.7475	31.7655	0.9196791	OTJPMultiLists
SFTwigStack	OTJPMultiLists	OXQ4	5.59E-39	TRUE	510.586	1.052	0.9197224	OTJPMultiLists
SFTwigStack	OTJPMultiLists	OXQ5	2.56E-34	TRUE	49.549	22.0235	0.859788	OTJPMultiLists
SFTwigStackList	SFTwigStackPrime	OXQ1	1.73E-04	TRUE	35.3105	35.1695	0.253018	SFTwigStackPrime
SFTwigStackList	SFTwigStackPrime	OXQ2	1.09E-32	TRUE	8.5255	8.805	0.8378398	SFTwigStackList
SFTwigStackList	SFTwigStackPrime	OXQ3	3.52E-45	TRUE	639.935	628.899	0.9939336	SFTwigStackPrime
SFTwigStackList	SFTwigStackPrime	OXQ4	3.52E-45	TRUE	516.767	517.802	0.9939336	SFTwigStackList
SFTwigStackList	SFTwigStackPrime	OXQ5	7.55E-01	FALSE	45.47	45.5235	0.04890127	tie
SFTwigStackList	OTJPrime	OXQ1	2.56E-34	TRUE	35.3105	18.1765	0.8597961	OTJPrime
SFTwigStackList	OTJPrime	OXQ2	2.55E-34	TRUE	8.5255	3.9685	0.8598049	OTJPrime
SFTwigStackList	OTJPrime	OXQ3	5.64E-39	TRUE	639.935	77.632	0.9196779	OTJPrime
SFTwigStackList	OTJPrime	OXQ4	5.64E-39	TRUE	516.767	17.0415	0.9196787	OTJPrime
SFTwigStackList	OTJPrime	OXQ5	2.56E-34	TRUE	45.47	24.612	0.859788	OTJPrime
SFTwigStackList	OTJPrimeList	OXQ1	2.56E-34	TRUE	35.3105	14.0375	0.8597958	OTJPrimeList
SFTwigStackList	OTJPrimeList	OXQ2	2.56E-34	TRUE	8.5255	2.891	0.8598023	OTJPrimeList
SFTwigStackList	OTJPrimeList	OXQ3	5.64E-39	TRUE	639.935	31.3805	0.9196791	OTJPrimeList
SFTwigStackList	OTJPrimeList	OXQ4	5.60E-39	TRUE	516.767	1.003	0.9197196	OTJPrimeList
SFTwigStackList	OTJPrimeList	OXQ5	2.56E-34	TRUE	45.47	21.9755	0.859788	OTJPrimeList
SFTwigStackList	OTJPMultiLists	OXQ1	2.56E-34	TRUE	35.3105	14.204	0.8597958	OTJPMultiLists
SFTwigStackList	OTJPMultiLists	OXQ2	2.55E-34	TRUE	8.5255	2.996	0.859815	OTJPMultiLists
SFTwigStackList	OTJPMultiLists	OXQ3	5.64E-39	TRUE	639.935	31.7655	0.9196791	OTJPMultiLists
SFTwigStackList	OTJPMultiLists	OXQ4	5.59E-39	TRUE	516.767	1.052	0.9197224	OTJPMultiLists
SFTwigStackList	OTJPMultiLists	OXQ5	2.56E-34	TRUE	45.47	22.0235	0.8597883	OTJPMultiLists
SFTwigStackPrime	OTJPrime	OXQ1	2.56E-34	TRUE	35.1695	18.1765	0.8597961	OTJPrime
SFTwigStackPrime	OTJPrime	OXQ2	2.55E-34	TRUE	8.805	3.9685	0.8598059	OTJPrime
SFTwigStackPrime	OTJPrime	OXQ3	5.64E-39	TRUE	628.899	77.632	0.9196779	OTJPrime
SFTwigStackPrime	OTJPrime	OXQ4	5.64E-39	TRUE	517.802	17.0415	0.9196787	OTJPrime
SFTwigStackPrime	OTJPrime	OXQ5	2.56E-34	TRUE	45.5235	24.612	0.859788	OTJPrime

Table B.4: Results for paired comparisons based on the U test over the XMark dataset.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	Median A	Median B	effect size	Winner
SFTwigStackPrime	OTJPrimeList	OXQ1	2.56E-34	TRUE	35.1695	14.0375	0.8597958	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	OXQ2	2.55E-34	TRUE	8.805	2.891	0.8598033	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	OXQ3	5.64E-39	TRUE	628.899	31.3805	0.9196791	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	OXQ4	5.60E-39	TRUE	517.802	1.003	0.9197196	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	OXQ5	2.56E-34	TRUE	45.5235	21.9755	0.859788	OTJPrimeList
SFTwigStackPrime	OTJPMultiLists	OXQ1	2.56E-34	TRUE	35.1695	14.204	0.8597958	OTJPMultiLists
SFTwigStackPrime	OTJPMultiLists	OXQ2	2.55E-34	TRUE	8.805	2.996	0.859816	OTJPMultiLists
SFTwigStackPrime	OTJPMultiLists	OXQ3	5.64E-39	TRUE	628.899	31.7655	0.9196791	OTJPMultiLists
SFTwigStackPrime	OTJPMultiLists	OXQ4	5.59E-39	TRUE	517.802	1.052	0.9197224	OTJPMultiLists
SFTwigStackPrime	OTJPMultiLists	OXQ5	2.56E-34	TRUE	45.5235	22.0235	0.8597883	OTJPMultiLists
OTJPrime	OTJPrimeList	OXQ1	2.56E-34	TRUE	18.1765	14.0375	0.8598003	OTJPrimeList
OTJPrime	OTJPrimeList	OXQ2	2.55E-34	TRUE	3.9685	2.891	0.8598098	OTJPrimeList
OTJPrime	OTJPrimeList	OXQ3	2.56E-34	TRUE	77.632	31.3805	0.859788	OTJPrimeList
OTJPrime	OTJPrimeList	OXQ4	2.55E-34	TRUE	17.0415	1.003	0.8598218	OTJPrimeList
OTJPrime	OTJPrimeList	OXQ5	1.60E-26	TRUE	24.612	21.9755	0.7490674	OTJPrimeList
OTJPrime	OTJPMultiLists	OXQ1	2.56E-34	TRUE	18.1765	14.204	0.8598003	OTJPMultiLists
OTJPrime	OTJPMultiLists	OXQ2	2.55E-34	TRUE	3.9685	2.996	0.8598225	OTJPMultiLists
OTJPrime	OTJPMultiLists	OXQ3	2.56E-34	TRUE	77.632	31.7655	0.859788	OTJPMultiLists
OTJPrime	OTJPMultiLists	OXQ4	2.55E-34	TRUE	17.0415	1.052	0.8598241	OTJPMultiLists
OTJPrime	OTJPMultiLists	OXQ5	1.08E-30	TRUE	24.612	22.0235	0.8101351	OTJPMultiLists
OTJPrimeList	OTJPMultiLists	OXQ1	1.49E-22	TRUE	14.0375	14.204	0.6859546	OTJPrimeList
OTJPrimeList	OTJPMultiLists	OXQ2	2.91E-25	TRUE	2.891	2.996	0.7296294	OTJPrimeList
OTJPrimeList	OTJPMultiLists	OXQ3	8.30E-10	TRUE	31.3805	31.7655	0.4262409	OTJPrimeList
OTJPrimeList	OTJPMultiLists	OXQ4	8.85E-17	TRUE	1.003	1.052	0.5824281	OTJPrimeList
OTJPrimeList	OTJPMultiLists	OXQ5	6.70E-01	FALSE	21.9755	22.0235	0.03107458	tie

Table B.5: Results for paired comparisons based on the U test over the TreeBank dataset.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	Median A	Median B	effect size	Winner
SFTwigStack	SFTwigStackList	OTQ1	0.09636652	FALSE	61.2015	61.066	0.09210324	tie
SFTwigStack	SFTwigStackList	OTQ2	3.52E-45	TRUE	1276.887	1223.651	0.9939336	SFTwigStackList
SFTwigStack	SFTwigStackList	OTQ3	4.91E-33	TRUE	0.527	0.493	0.8425519	SFTwigStackList
SFTwigStack	SFTwigStackList	OTQ4	3.52E-45	TRUE	1229.29	1178.948	0.9939336	SFTwigStackList
SFTwigStack	SFTwigStackList	OTQ5	3.52E-45	TRUE	73.272	73.798	0.9939336	SFTwigStack
SFTwigStack	SFTwigStackList	OTQ6	2.50E-34	TRUE	1.449	0.9295	0.8599322	SFTwigStackList
SFTwigStack	SFTwigStackList	OTQ7	3.52E-45	TRUE	70.312	74.951	0.9939336	SFTwigStack
SFTwigStack	SFTwigStackList	OTQ8	3.52E-45	TRUE	103.697	103.022	0.9939336	SFTwigStackList
SFTwigStack	SFTwigStackList	OTQ9	3.87E-04	TRUE	0.117	0.12	0.2377369	SFTwigStack
SFTwigStack	SFTwigStackPrime	OTQ1	3.96E-33	TRUE	61.2015	55.3235	0.8438185	SFTwigStackPrime
SFTwigStack	SFTwigStackPrime	OTQ2	3.52E-45	TRUE	1276.887	878.387	0.9939336	SFTwigStackPrime
SFTwigStack	SFTwigStackPrime	OTQ3	2.48E-34	TRUE	0.527	0.757	0.8599699	SFTwigStack
SFTwigStack	SFTwigStackPrime	OTQ4	3.52E-45	TRUE	1229.29	888.492	0.9939336	SFTwigStackPrime
SFTwigStack	SFTwigStackPrime	OTQ5	3.52E-45	TRUE	73.272	73.548	0.9939336	SFTwigStack
SFTwigStack	SFTwigStackPrime	OTQ6	2.53E-34	TRUE	1.449	1.3405	0.8598703	SFTwigStackPrime
SFTwigStack	SFTwigStackPrime	OTQ7	3.52E-45	TRUE	70.312	74.664	0.9939336	SFTwigStack
SFTwigStack	SFTwigStackPrime	OTQ8	3.52E-45	TRUE	103.697	102.912	0.9939336	SFTwigStackPrime
SFTwigStack	SFTwigStackPrime	OTQ9	0.01032286	TRUE	0.117	0.119	0.1636529	SFTwigStack
SFTwigStack	OTJPrime	OTQ1	4.49E-31	TRUE	61.2015	56.3025	0.8155185	OTJPrime

Table B.5: Results for paired comparisons based on the U test over the TreeBank dataset.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	Median A	Median B	effect size	Winner
SFTwigStack	OTJPrime	OTQ2	3.52E-45	TRUE	1276.887	889.6	0.9939336	OTJPrime
SFTwigStack	OTJPrime	OTQ3	2.47E-34	TRUE	0.527	0.7115	0.8600054	SFTwigStack
SFTwigStack	OTJPrime	OTQ4	5.62E-39	TRUE	1229.29	4.9345	0.9196981	OTJPrime
SFTwigStack	OTJPrime	OTQ5	5.60E-39	TRUE	73.272	1.869	0.9197152	OTJPrime
SFTwigStack	OTJPrime	OTQ6	1.53E-22	TRUE	1.449	1.424	0.6858001	OTJPrime
SFTwigStack	OTJPrime	OTQ7	5.57E-39	TRUE	70.312	1.853	0.9197423	OTJPrime
SFTwigStack	OTJPrime	OTQ8	5.64E-39	TRUE	103.697	42.9305	0.9196783	OTJPrime
SFTwigStack	OTJPrime	OTQ9	2.15E-01	FALSE	0.117	0.1165	0.05569781	tie
SFTwigStack	OTJPrimeList	OTQ1	2.56E-34	TRUE	61.2015	21.5905	0.859789	OTJPrimeList
SFTwigStack	OTJPrimeList	OTQ2	3.52E-45	TRUE	1276.887	890.809	0.9939336	OTJPrimeList
SFTwigStack	OTJPrimeList	OTQ3	6.37E-33	TRUE	0.527	0.558	0.8410178	SFTwigStack
SFTwigStack	OTJPrimeList	OTQ4	5.59E-39	TRUE	1229.29	1.5915	0.9197244	OTJPrimeList
SFTwigStack	OTJPrimeList	OTQ5	5.56E-39	TRUE	73.272	1.0295	0.9197542	OTJPrimeList
SFTwigStack	OTJPrimeList	OTQ6	2.52E-34	TRUE	1.449	1.064	0.8598915	OTJPrimeList
SFTwigStack	OTJPrimeList	OTQ7	5.60E-39	TRUE	70.312	1.8715	0.9197164	OTJPrimeList
SFTwigStack	OTJPrimeList	OTQ8	5.64E-39	TRUE	103.697	30.401	0.9196791	OTJPrimeList
SFTwigStack	OTJPrimeList	OTQ9	1.02E-08	TRUE	0.117	0.122	0.3965471	SFTwigStack
SFTwigStack	OTJPMultiLists	OTQ1	2.56E-34	TRUE	61.2015	22.87	0.8597919	OTJPMultiLists
SFTwigStack	OTJPMultiLists	OTQ2	3.52E-45	TRUE	1276.887	910.661	0.9939336	OTJPMultiLists
SFTwigStack	OTJPMultiLists	OTQ3	2.47E-34	TRUE	0.527	0.739	0.8599882	SFTwigStack
SFTwigStack	OTJPMultiLists	OTQ4	5.59E-39	TRUE	1229.29	2.2595	0.9197212	OTJPMultiLists
SFTwigStack	OTJPMultiLists	OTQ5	5.55E-39	TRUE	73.272	1.057	0.9197609	OTJPMultiLists
SFTwigStack	OTJPMultiLists	OTQ6	3.78E-10	TRUE	1.449	1.464	0.4351392	SFTwigStack
SFTwigStack	OTJPMultiLists	OTQ7	5.59E-39	TRUE	70.312	1.8895	0.9197264	OTJPMultiLists
SFTwigStack	OTJPMultiLists	OTQ8	5.64E-39	TRUE	103.697	30.508	0.9196787	OTJPMultiLists
SFTwigStack	OTJPMultiLists	OTQ9	6.99E-11	TRUE	0.117	0.123	0.4536918	SFTwigStack
SFTwigStackList	SFTwigStackPrime	OTQ1	9.53E-32	TRUE	61.066	55.3235	0.824894	SFTwigStackPrime
SFTwigStackList	SFTwigStackPrime	OTQ2	3.52E-45	TRUE	1223.651	878.387	0.9939336	SFTwigStackPrime
SFTwigStackList	SFTwigStackPrime	OTQ3	2.46E-34	TRUE	0.493	0.757	0.8600253	SFTwigStackList
SFTwigStackList	SFTwigStackPrime	OTQ4	3.52E-45	TRUE	1178.948	888.492	0.9939336	SFTwigStackPrime
SFTwigStackList	SFTwigStackPrime	OTQ5	3.52E-45	TRUE	73.798	73.548	0.9939336	SFTwigStackPrime
SFTwigStackList	SFTwigStackPrime	OTQ6	2.50E-34	TRUE	0.9295	1.3405	0.8599289	SFTwigStackList
SFTwigStackList	SFTwigStackPrime	OTQ7	3.52E-45	TRUE	74.951	74.664	0.9939336	SFTwigStackPrime
SFTwigStackList	SFTwigStackPrime	OTQ8	3.52E-45	TRUE	103.022	102.912	0.9939336	SFTwigStackPrime
SFTwigStackList	SFTwigStackPrime	OTQ9	2.49E-01	FALSE	0.12	0.119	0.04785163	tie
SFTwigStackList	OTJPrime	OTQ1	1.97E-29	TRUE	61.066	56.3025	0.7921585	OTJPrime
SFTwigStackList	OTJPrime	OTQ2	3.52E-45	TRUE	1223.651	889.6	0.9939336	OTJPrime
SFTwigStackList	OTJPrime	OTQ3	2.44E-34	TRUE	0.493	0.7115	0.8600608	SFTwigStackList
SFTwigStackList	OTJPrime	OTQ4	5.62E-39	TRUE	1178.948	4.9345	0.9196981	OTJPrime
SFTwigStackList	OTJPrime	OTQ5	5.60E-39	TRUE	73.798	1.869	0.9197152	OTJPrime
SFTwigStackList	OTJPrime	OTQ6	2.49E-34	TRUE	0.9295	1.424	0.8599478	SFTwigStackList
SFTwigStackList	OTJPrime	OTQ7	5.57E-39	TRUE	74.951	1.853	0.9197423	OTJPrime
SFTwigStackList	OTJPrime	OTQ8	5.64E-39	TRUE	103.022	42.9305	0.9196783	OTJPrime
SFTwigStackList	OTJPrime	OTQ9	4.03E-06	TRUE	0.12	0.1165	0.3156219	OTJPrime
SFTwigStackList	OTJPrimeList	OTQ1	2.56E-34	TRUE	61.066	21.5905	0.8597883	OTJPrimeList
SFTwigStackList	OTJPrimeList	OTQ2	3.52E-45	TRUE	1223.651	890.809	0.9939336	OTJPrimeList
SFTwigStackList	OTJPrimeList	OTQ3	2.44E-34	TRUE	0.493	0.558	0.8600663	SFTwigStackList
SFTwigStackList	OTJPrimeList	OTQ4	5.59E-39	TRUE	1178.948	1.5915	0.9197244	OTJPrimeList

Table B.5: Results for paired comparisons based on the U test over the TreeBank dataset.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	Median A	Median B	effect size	Winner
SFTwigStackList	OTJPrimeList	OTQ5	5.56E-39	TRUE	73.798	1.0295	0.9197542	OTJPrimeList
SFTwigStackList	OTJPrimeList	OTQ6	2.49E-34	TRUE	0.9295	1.064	0.8599501	SFTwigStackList
SFTwigStackList	OTJPrimeList	OTQ7	5.60E-39	TRUE	74.951	1.8715	0.9197164	OTJPrimeList
SFTwigStackList	OTJPrimeList	OTQ8	5.64E-39	TRUE	103.022	30.401	0.9196791	OTJPrimeList
SFTwigStackList	OTJPrimeList	OTQ9	1.84E-02	TRUE	0.12	0.122	0.1476047	SFTwigStackList
SFTwigStackList	OTJPMultiLists	OTQ1	2.56E-34	TRUE	61.066	22.87	0.8597912	OTJPMultiLists
SFTwigStackList	OTJPMultiLists	OTQ2	3.52E-45	TRUE	1223.651	910.661	0.9939336	OTJPMultiLists
SFTwigStackList	OTJPMultiLists	OTQ3	2.45E-34	TRUE	0.493	0.739	0.8600435	SFTwigStackList
SFTwigStackList	OTJPMultiLists	OTQ4	5.59E-39	TRUE	1178.948	2.2595	0.9197212	OTJPMultiLists
SFTwigStackList	OTJPMultiLists	OTQ5	5.55E-39	TRUE	73.798	1.057	0.9197609	OTJPMultiLists
SFTwigStackList	OTJPMultiLists	OTQ6	2.49E-34	TRUE	0.9295	1.464	0.8599475	SFTwigStackList
SFTwigStackList	OTJPMultiLists	OTQ7	5.59E-39	TRUE	74.951	1.8895	0.9197264	OTJPMultiLists
SFTwigStackList	OTJPMultiLists	OTQ8	5.64E-39	TRUE	103.022	30.508	0.9196787	OTJPMultiLists
SFTwigStackList	OTJPMultiLists	OTQ9	8.02E-04	TRUE	0.12	0.123	0.2230932	SFTwigStackList
SFTwigStackPrime	OTJPrime	OTQ1	2.32E-07	TRUE	55.3235	56.3025	0.3564094	SFTwigStackPrime
SFTwigStackPrime	OTJPrime	OTQ2	3.52E-45	TRUE	878.387	889.6	0.9939336	SFTwigStackPrime
SFTwigStackPrime	OTJPrime	OTQ3	7.53E-34	TRUE	0.757	0.7115	0.8535377	OTJPrime
SFTwigStackPrime	OTJPrime	OTQ4	5.62E-39	TRUE	888.492	4.9345	0.9196981	OTJPrime
SFTwigStackPrime	OTJPrime	OTQ5	5.60E-39	TRUE	73.548	1.869	0.9197152	OTJPrime
SFTwigStackPrime	OTJPrime	OTQ6	4.80E-34	TRUE	1.3405	1.424	0.8561541	SFTwigStackPrime
SFTwigStackPrime	OTJPrime	OTQ7	5.57E-39	TRUE	74.664	1.853	0.9197423	OTJPrime
SFTwigStackPrime	OTJPrime	OTQ8	5.64E-39	TRUE	102.912	42.9305	0.9196783	OTJPrime
SFTwigStackPrime	OTJPrime	OTQ9	0.000177207	TRUE	0.119	0.1165	0.2525708	OTJPrime
SFTwigStackPrime	OTJPrimeList	OTQ1	2.56E-34	TRUE	55.3235	21.5905	0.859788	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	OTQ2	3.52E-45	TRUE	878.387	890.809	0.9939336	SFTwigStackPrime
SFTwigStackPrime	OTJPrimeList	OTQ3	2.49E-34	TRUE	0.757	0.558	0.8599634	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	OTQ4	5.59E-39	TRUE	888.492	1.5915	0.9197244	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	OTQ5	5.56E-39	TRUE	73.548	1.0295	0.9197542	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	OTQ6	2.52E-34	TRUE	1.3405	1.064	0.8598882	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	OTQ7	5.60E-39	TRUE	74.664	1.8715	0.9197164	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	OTQ8	5.64E-39	TRUE	102.912	30.401	0.9196791	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	OTQ9	1.85E-04	TRUE	0.119	0.122	0.2517365	SFTwigStackPrime
SFTwigStackPrime	OTJPMultiLists	OTQ1	2.56E-34	TRUE	55.3235	22.87	0.8597909	OTJPMultiLists
SFTwigStackPrime	OTJPMultiLists	OTQ2	3.52E-45	TRUE	878.387	910.661	0.9939336	SFTwigStackPrime
SFTwigStackPrime	OTJPMultiLists	OTQ3	2.12E-19	TRUE	0.757	0.739	0.6314879	OTJPMultiLists
SFTwigStackPrime	OTJPMultiLists	OTQ4	5.59E-39	TRUE	888.492	2.2595	0.9197212	OTJPMultiLists
SFTwigStackPrime	OTJPMultiLists	OTQ5	5.55E-39	TRUE	73.548	1.057	0.9197609	OTJPMultiLists
SFTwigStackPrime	OTJPMultiLists	OTQ6	2.52E-34	TRUE	1.3405	1.464	0.8598856	SFTwigStackPrime
SFTwigStackPrime	OTJPMultiLists	OTQ7	5.59E-39	TRUE	74.664	1.8895	0.9197264	OTJPMultiLists
SFTwigStackPrime	OTJPMultiLists	OTQ8	5.64E-39	TRUE	102.912	30.508	0.9196787	OTJPMultiLists
SFTwigStackPrime	OTJPMultiLists	OTQ9	1.04E-06	TRUE	0.119	0.123	0.3355727	SFTwigStackPrime
OTJPrime	OTJPrimeList	OTQ1	2.56E-34	TRUE	56.3025	21.5905	0.8597886	OTJPrimeList
OTJPrime	OTJPrimeList	OTQ2	3.52E-45	TRUE	889.6	890.809	0.9939336	OTJPrime
OTJPrime	OTJPrimeList	OTQ3	2.47E-34	TRUE	0.7115	0.558	0.8599989	OTJPrimeList
OTJPrime	OTJPrimeList	OTQ4	2.54E-34	TRUE	4.9345	1.5915	0.8598417	OTJPrimeList
OTJPrime	OTJPrimeList	OTQ5	2.52E-34	TRUE	1.869	1.0295	0.8598801	OTJPrimeList
OTJPrime	OTJPrimeList	OTQ6	2.51E-34	TRUE	1.424	1.064	0.8599071	OTJPrimeList
OTJPrime	OTJPrimeList	OTQ7	1.44E-07	TRUE	1.853	1.8715	0.3628495	OTJPrime

Table B.5: Results for paired comparisons based on the U test over the TreeBank dataset.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	Median A	Median B	effect size	Winner
OTJPrime	OTJPrimeList	OTQ8	1.91E-28	TRUE	42.9305	30.401	0.7778266	OTJPrimeList
OTJPrime	OTJPrimeList	OTQ9	3.88E-11	TRUE	0.1165	0.122	0.459995	OTJPrime
OTJPrime	OTJPMultiLists	OTQ1	2.56E-34	TRUE	56.3025	22.87	0.8597916	OTJPMultiLists
OTJPrime	OTJPMultiLists	OTQ2	3.52E-45	TRUE	889.6	910.661	0.9939336	OTJPrime
OTJPrime	OTJPMultiLists	OTQ3	9.22E-31	TRUE	0.7115	0.739	0.8111244	OTJPrime
OTJPrime	OTJPMultiLists	OTQ4	2.54E-34	TRUE	4.9345	2.2595	0.8598391	OTJPMultiLists
OTJPrime	OTJPMultiLists	OTQ5	4.79E-33	TRUE	1.869	1.057	0.8426991	OTJPMultiLists
OTJPrime	OTJPMultiLists	OTQ6	9.45E-29	TRUE	1.424	1.464	0.7822915	OTJPrime
OTJPrime	OTJPMultiLists	OTQ7	9.88E-19	TRUE	1.853	1.8895	0.6193305	OTJPrime
OTJPrime	OTJPMultiLists	OTQ8	1.60E-29	TRUE	42.9305	30.508	0.793461	OTJPMultiLists
OTJPrime	OTJPMultiLists	OTQ9	2.63E-13	TRUE	0.1165	0.123	0.5104265	OTJPrime
OTJPrimeList	OTJPMultiLists	OTQ1	1.21E-26	TRUE	21.5905	22.87	0.7508958	OTJPrimeList
OTJPrimeList	OTJPMultiLists	OTQ2	3.52E-45	TRUE	890.809	910.661	0.9939336	OTJPrimeList
OTJPrimeList	OTJPMultiLists	OTQ3	2.48E-34	TRUE	0.558	0.739	0.8599817	OTJPrimeList
OTJPrimeList	OTJPMultiLists	OTQ4	2.53E-34	TRUE	1.5915	2.2595	0.8598606	OTJPrimeList
OTJPrimeList	OTJPMultiLists	OTQ5	9.27E-23	TRUE	1.0295	1.057	0.6893907	OTJPrimeList
OTJPrimeList	OTJPMultiLists	OTQ6	2.51E-34	TRUE	1.064	1.464	0.8599068	OTJPrimeList
OTJPrimeList	OTJPMultiLists	OTQ7	6.07E-08	TRUE	1.8715	1.8895	0.3741665	OTJPrimeList
OTJPrimeList	OTJPMultiLists	OTQ8	8.63E-02	FALSE	30.401	30.508	0.09644356	tie
OTJPrimeList	OTJPMultiLists	OTQ9	1.96E-01	FALSE	0.122	0.123	0.06053243	tie

Table B.6: Results for paired comparisons based on the U test over the Random dataset.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	Median A	Median B	effect size	Winner
SFTwigStack	SFTwigStackList	ORQ1	3.52E-45	TRUE	7.856	8.497	0.9939336	SFTwigStack
SFTwigStack	SFTwigStackList	ORQ2	3.52E-45	TRUE	879.209	899.929	0.9939336	SFTwigStack
SFTwigStack	SFTwigStackList	ORQ3	2.15E-37	TRUE	104.337	107.353	0.8997745	SFTwigStack
SFTwigStack	SFTwigStackList	ORQ4	3.52E-45	TRUE	200.485	201.732	0.9939336	SFTwigStack
SFTwigStack	SFTwigStackList	ORQ5	7.68E-43	TRUE	192.686	100.157	0.9666109	SFTwigStackList
SFTwigStack	SFTwigStackList	ORQ6	3.52E-45	TRUE	2718.407	179.706	0.9939336	SFTwigStackList
SFTwigStack	SFTwigStackList	ORQ7	2.55E-34	TRUE	14.529	9.1885	0.8598095	SFTwigStackList
SFTwigStack	SFTwigStackPrime	ORQ1	3.52E-45	TRUE	7.856	8.095	0.9939336	SFTwigStack
SFTwigStack	SFTwigStackPrime	ORQ2	3.52E-45	TRUE	879.209	898.262	0.9939336	SFTwigStack
SFTwigStack	SFTwigStackPrime	ORQ3	2.15E-37	TRUE	104.337	117.697	0.8997745	SFTwigStack
SFTwigStack	SFTwigStackPrime	ORQ4	3.52E-45	TRUE	200.485	200.692	0.9939336	SFTwigStack
SFTwigStack	SFTwigStackPrime	ORQ5	5.76E-45	TRUE	192.686	113.055	0.9914657	SFTwigStackPrime
SFTwigStack	SFTwigStackPrime	ORQ6	3.52E-45	TRUE	2718.407	136.726	0.9939336	SFTwigStackPrime
SFTwigStack	SFTwigStackPrime	ORQ7	2.55E-34	TRUE	14.529	10.442	0.8598075	SFTwigStackPrime
SFTwigStack	OTJPrime	ORQ1	5.63E-39	TRUE	7.856	6.304	0.9196854	OTJPrime
SFTwigStack	OTJPrime	ORQ2	3.52E-45	TRUE	879.209	184.288	0.9939336	OTJPrime
SFTwigStack	OTJPrime	ORQ3	2.59E-38	TRUE	104.337	43.0315	0.9114022	OTJPrime
SFTwigStack	OTJPrime	ORQ4	2.65E-43	TRUE	200.485	113.413	0.9720684	OTJPrime
SFTwigStack	OTJPrime	ORQ5	1.82E-41	TRUE	192.686	105.155	0.9501823	OTJPrime
SFTwigStack	OTJPrime	ORQ6	3.52E-45	TRUE	2718.407	113.088	0.9939336	OTJPrime
SFTwigStack	OTJPrime	ORQ7	2.55E-34	TRUE	14.529	9.167	0.8598082	OTJPrime
SFTwigStack	OTJPrimeList	ORQ1	4.69E-36	TRUE	7.856	6.4575	0.8825932	OTJPrimeList
SFTwigStack	OTJPrimeList	ORQ2	5.63E-39	TRUE	879.209	10.462	0.9196918	OTJPrimeList
SFTwigStack	OTJPrimeList	ORQ3	2.26E-38	TRUE	104.337	10.2265	0.912138	OTJPrimeList

Table B.6: Results for paired comparisons based on the U test over the Random dataset.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	Median A	Median B	effect size	Winner
SFTwigStack	OTJPrimeList	ORQ4	5.64E-39	TRUE	200.485	14.5015	0.9196783	OTJPrimeList
SFTwigStack	OTJPrimeList	ORQ5	1.42E-41	TRUE	192.686	101.964	0.9514658	OTJPrimeList
SFTwigStack	OTJPrimeList	ORQ6	3.52E-45	TRUE	2718.407	123.718	0.9939336	OTJPrimeList
SFTwigStack	OTJPrimeList	ORQ7	2.55E-34	TRUE	14.529	8.3745	0.8598114	OTJPrimeList
SFTwigStack	OTJPMultiLists	ORQ1	4.70E-36	TRUE	7.856	6.633	0.8825894	OTJPMultiLists
SFTwigStack	OTJPMultiLists	ORQ2	5.63E-39	TRUE	879.209	10.689	0.9196902	OTJPMultiLists
SFTwigStack	OTJPMultiLists	ORQ3	2.26E-38	TRUE	104.337	11.318	0.9121388	OTJPMultiLists
SFTwigStack	OTJPMultiLists	ORQ4	5.64E-39	TRUE	200.485	14.8105	0.9196783	OTJPMultiLists
SFTwigStack	OTJPMultiLists	ORQ5	3.52E-45	TRUE	192.686	134.224	0.9939336	OTJPMultiLists
SFTwigStack	OTJPMultiLists	ORQ6	3.52E-45	TRUE	2718.407	115.325	0.9939336	OTJPMultiLists
SFTwigStack	OTJPMultiLists	ORQ7	2.55E-34	TRUE	14.529	9.446	0.8598059	OTJPMultiLists
SFTwigStackList	SFTwigStackPrime	ORQ1	3.52E-45	TRUE	8.497	8.095	0.9939336	SFTwigStackPrime
SFTwigStackList	SFTwigStackPrime	ORQ2	3.52E-45	TRUE	899.929	898.262	0.9939336	SFTwigStackPrime
SFTwigStackList	SFTwigStackPrime	ORQ3	3.54E-37	TRUE	107.353	117.697	0.8970221	SFTwigStackList
SFTwigStackList	SFTwigStackPrime	ORQ4	3.52E-45	TRUE	201.732	200.692	0.9939336	SFTwigStackPrime
SFTwigStackList	SFTwigStackPrime	ORQ5	3.31E-41	TRUE	100.157	113.055	0.947044	SFTwigStackList
SFTwigStackList	SFTwigStackPrime	ORQ6	3.52E-45	TRUE	179.706	136.726	0.9939336	SFTwigStackPrime
SFTwigStackList	SFTwigStackPrime	ORQ7	2.55E-34	TRUE	9.1885	10.442	0.8598072	SFTwigStackList
SFTwigStackList	OTJPrime	ORQ1	5.63E-39	TRUE	8.497	6.304	0.9196854	OTJPrime
SFTwigStackList	OTJPrime	ORQ2	3.52E-45	TRUE	899.929	184.288	0.9939336	OTJPrime
SFTwigStackList	OTJPrime	ORQ3	2.59E-38	TRUE	107.353	43.0315	0.9114022	OTJPrime
SFTwigStackList	OTJPrime	ORQ4	2.65E-43	TRUE	201.732	113.413	0.9720684	OTJPrime
SFTwigStackList	OTJPrime	ORQ5	1.65E-13	TRUE	100.157	105.155	0.5148825	SFTwigStackList
SFTwigStackList	OTJPrime	ORQ6	3.52E-45	TRUE	179.706	113.088	0.9939336	OTJPrime
SFTwigStackList	OTJPrime	ORQ7	0.000326623	TRUE	9.1885	9.167	0.2410163	OTJPrime
SFTwigStackList	OTJPrimeList	ORQ1	5.63E-39	TRUE	8.497	6.4575	0.9196886	OTJPrimeList
SFTwigStackList	OTJPrimeList	ORQ2	5.63E-39	TRUE	899.929	10.462	0.9196918	OTJPrimeList
SFTwigStackList	OTJPrimeList	ORQ3	2.26E-38	TRUE	107.353	10.2265	0.912138	OTJPrimeList
SFTwigStackList	OTJPrimeList	ORQ4	5.64E-39	TRUE	201.732	14.5015	0.9196783	OTJPrimeList
SFTwigStackList	OTJPrimeList	ORQ5	4.35E-13	TRUE	100.157	101.964	0.5055609	SFTwigStackList
SFTwigStackList	OTJPrimeList	ORQ6	3.52E-45	TRUE	179.706	123.718	0.9939336	OTJPrimeList
SFTwigStackList	OTJPrimeList	ORQ7	4.71E-33	TRUE	9.1885	8.3745	0.8427996	OTJPrimeList
SFTwigStackList	OTJPMultiLists	ORQ1	1.68E-37	TRUE	8.497	6.633	0.9011386	OTJPMultiLists
SFTwigStackList	OTJPMultiLists	ORQ2	5.63E-39	TRUE	899.929	10.689	0.9196902	OTJPMultiLists
SFTwigStackList	OTJPMultiLists	ORQ3	2.26E-38	TRUE	107.353	11.318	0.9121388	OTJPMultiLists
SFTwigStackList	OTJPMultiLists	ORQ4	5.64E-39	TRUE	201.732	14.8105	0.9196783	OTJPMultiLists
SFTwigStackList	OTJPMultiLists	ORQ5	7.68E-43	TRUE	100.157	134.224	0.9666109	SFTwigStackList
SFTwigStackList	OTJPMultiLists	ORQ6	3.52E-45	TRUE	179.706	115.325	0.9939336	OTJPMultiLists
SFTwigStackList	OTJPMultiLists	ORQ7	6.11E-29	TRUE	9.1885	9.446	0.7850524	SFTwigStackList
SFTwigStackPrime	OTJPrime	ORQ1	5.63E-39	TRUE	8.095	6.304	0.9196854	OTJPrime
SFTwigStackPrime	OTJPrime	ORQ2	3.52E-45	TRUE	898.262	184.288	0.9939336	OTJPrime
SFTwigStackPrime	OTJPrime	ORQ3	2.59E-38	TRUE	117.697	43.0315	0.9114022	OTJPrime
SFTwigStackPrime	OTJPrime	ORQ4	2.65E-43	TRUE	200.692	113.413	0.9720684	OTJPrime
SFTwigStackPrime	OTJPrime	ORQ5	6.36E-40	TRUE	113.055	105.155	0.9313985	OTJPrime
SFTwigStackPrime	OTJPrime	ORQ6	3.52E-45	TRUE	136.726	113.088	0.9939336	OTJPrime
SFTwigStackPrime	OTJPrime	ORQ7	2.55E-34	TRUE	10.442	9.167	0.8598059	OTJPrime
SFTwigStackPrime	OTJPrimeList	ORQ1	4.69E-36	TRUE	8.095	6.4575	0.8825932	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	ORQ2	5.63E-39	TRUE	898.262	10.462	0.9196918	OTJPrimeList

Table B.6: Results for paired comparisons based on the U test over the Random dataset.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	Median A	Median B	effect size	Winner
SFTwigStackPrime	OTJPrimeList	ORQ3	2.26E-38	TRUE	117.697	10.2265	0.912138	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	ORQ4	5.64E-39	TRUE	200.692	14.5015	0.9196783	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	ORQ5	3.51E-40	TRUE	113.055	101.964	0.9345648	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	ORQ6	3.52E-45	TRUE	136.726	123.718	0.9939336	OTJPrimeList
SFTwigStackPrime	OTJPrimeList	ORQ7	2.55E-34	TRUE	10.442	8.3745	0.8598091	OTJPrimeList
SFTwigStackPrime	OTJPMultiLists	ORQ1	1.68E-37	TRUE	8.095	6.633	0.9011386	OTJPMultiLists
SFTwigStackPrime	OTJPMultiLists	ORQ2	5.63E-39	TRUE	898.262	10.689	0.9196902	OTJPMultiLists
SFTwigStackPrime	OTJPMultiLists	ORQ3	2.26E-38	TRUE	117.697	11.318	0.9121388	OTJPMultiLists
SFTwigStackPrime	OTJPMultiLists	ORQ4	5.64E-39	TRUE	200.692	14.8105	0.9196783	OTJPMultiLists
SFTwigStackPrime	OTJPMultiLists	ORQ5	5.76E-45	TRUE	113.055	134.224	0.9914657	SFTwigStackPrime
SFTwigStackPrime	OTJPMultiLists	ORQ6	3.52E-45	TRUE	136.726	115.325	0.9939336	OTJPMultiLists
SFTwigStackPrime	OTJPMultiLists	ORQ7	2.55E-34	TRUE	10.442	9.446	0.8598036	OTJPMultiLists
OTJPrime	OTJPrimeList	ORQ1	6.27E-12	TRUE	6.304	6.4575	0.4789848	OTJPrime
OTJPrime	OTJPrimeList	ORQ2	5.63E-39	TRUE	184.288	10.462	0.9196918	OTJPrimeList
OTJPrime	OTJPrimeList	ORQ3	2.56E-34	TRUE	43.0315	10.2265	0.8597877	OTJPrimeList
OTJPrime	OTJPrimeList	ORQ4	1.38E-37	TRUE	113.413	14.5015	0.9022356	OTJPrimeList
OTJPrime	OTJPrimeList	ORQ5	3.12E-15	TRUE	105.155	101.964	0.5514694	OTJPrimeList
OTJPrime	OTJPrimeList	ORQ6	3.52E-45	TRUE	113.088	123.718	0.9939336	OTJPrime
OTJPrime	OTJPrimeList	ORQ7	4.71E-33	TRUE	9.167	8.3745	0.8427983	OTJPrimeList
OTJPrime	OTJPMultiLists	ORQ1	1.23E-22	TRUE	6.304	6.633	0.6873409	OTJPrime
OTJPrime	OTJPMultiLists	ORQ2	5.63E-39	TRUE	184.288	10.689	0.9196902	OTJPMultiLists
OTJPrime	OTJPMultiLists	ORQ3	2.56E-34	TRUE	43.0315	11.318	0.8597883	OTJPMultiLists
OTJPrime	OTJPMultiLists	ORQ4	1.38E-37	TRUE	113.413	14.8105	0.9022356	OTJPMultiLists
OTJPrime	OTJPMultiLists	ORQ5	1.82E-41	TRUE	105.155	134.224	0.9501823	OTJPrime
OTJPrime	OTJPMultiLists	ORQ6	3.52E-45	TRUE	113.088	115.325	0.9939336	OTJPrime
OTJPrime	OTJPMultiLists	ORQ7	1.92E-31	TRUE	9.167	9.446	0.8206563	OTJPrime
OTJPrimeList	OTJPMultiLists	ORQ1	2.69E-14	TRUE	6.4575	6.633	0.5319174	OTJPrimeList
OTJPrimeList	OTJPMultiLists	ORQ2	6.27E-28	TRUE	10.462	10.689	0.7702004	OTJPrimeList
OTJPrimeList	OTJPMultiLists	ORQ3	1.77E-20	TRUE	10.2265	11.318	0.6506047	OTJPrimeList
OTJPrimeList	OTJPMultiLists	ORQ4	0.2651988	FALSE	14.5015	14.8105	0.04436382	tie
OTJPrimeList	OTJPMultiLists	ORQ5	1.42E-41	TRUE	101.964	134.224	0.9514658	OTJPrimeList
OTJPrimeList	OTJPMultiLists	ORQ6	3.52E-45	TRUE	123.718	115.325	0.9939336	OTJPMultiLists
OTJPrimeList	OTJPMultiLists	ORQ7	4.45E-33	TRUE	8.3745	9.446	0.8431433	OTJPrimeList

Table B.3: Experimental ordered TPQs for the Random dataset.

Code	XPath expression
ORQ_1	//b//e//a[/following::f][/following::d]
ORQ_2	//a//b[/following::e][/preceding-sibling::c]
ORQ_3	//a/c//b[/preceding-sibling::d][/preceding-sibling::e]
ORQ_4	//b//a//d[/following-sibling::f]/following-sibling::e
ORQ_5	//d[a//e/preceding-sibling::f]/c[b]
ORQ_6	//a[d][c][b][e]//f [d«c][c«b][b«e]
ORQ_7	//a[c//e]/f[d] [c«f]

Appendix C

Bottom-Up Holistic Approaches Full results

Bottom-Up Holistic Algorithms for $XPath^{/,//,[]}$

Table C.1: Experimental TPQs for the Zipf dataset.

Query	XPath expression	Result
ZQ1	//g/b[//d]/a	87116
ZQ2	//a/b[//d]/g	42886
ZQ3	//a/g[//b]/d	79146
ZQ4	//d/g[//a]/b	57184
ZQ5	//b/a[//g]/d	40200
ZQ6	//d/b[//a]/g	31161
ZQ7	//b/d[//g]/a	52463
ZQ8	//b/a[//d]/g	40224
ZQ9	//b/g[//a]/d	56263
ZQ10	//g/b[//a]/d	120316
ZQ11	//g[/c]/f	1672
ZQ12	//a[/c]/e	13775
ZQ13	//a[/c]/b	34673
ZQ14	//g[/d]/b	3758
ZQ15	//c[/g]/e	1997
ZQ16	//e[/c]/a	13849
ZQ17	//b[/c]/g	4942
ZQ18	//c[/a]/b	34877
ZQ19	//a[/f]/d	8763
ZQ20	//g[/f]/a	5062

Table C.1: Experimental TPQs for the Zipf dataset.

Query	XPath expression	Result
ZQ21	//g[/e][/f]//c	3267
ZQ22	//g[/a][/c]//b	78009
ZQ23	//a[/c][/b]//f	65035
ZQ24	//b[/a][/d]//f	57653
ZQ25	//b[/e][/d]//a	136372
ZQ26	//b[/c][/d]//f	21389
ZQ27	//a[/g][/c]//b	43298
ZQ28	//g[/e][/d]//f	2143
ZQ29	//a[/d][/g]//e	38488
ZQ30	//d[/c][/b]//g	10306
ZQ31	//b/a//d/g	20235
ZQ32	//g/a//d/b	24984
ZQ33	//d/g//a/b	19312
ZQ34	//a/g//d/b	31819
ZQ35	//d/g//b/a	19294
ZQ36	//b/d//a/g	21878
ZQ37	//a/d//b/g	18168
ZQ38	//d/b//g/a	17057
ZQ39	//b/d//g/a	21923
ZQ40	//g/d//b/a	26259
ZQ41	//g/g[/d]/b//b[/a]//c	38
ZQ42	//a/a[/g]/d//d[/b]//e	490
ZQ43	//b/b[/a]/d//d[/g]//e	206
ZQ44	//d/d[/a]/g//g[/b]//c	111
ZQ45	//d/d[/b]/g//g[/a]//e	53
ZQ46	//d/d[/b]/a//a[/g]//f	441
ZQ47	//a/a[/d]/b//b[/g]//f	748
ZQ48	//d/d[/b]/a//a[/g]//e	536
ZQ49	//d/d[/g]/b//b[/a]//f	79
ZQ50	//g/g[/a]/b//b[/d]//c	154

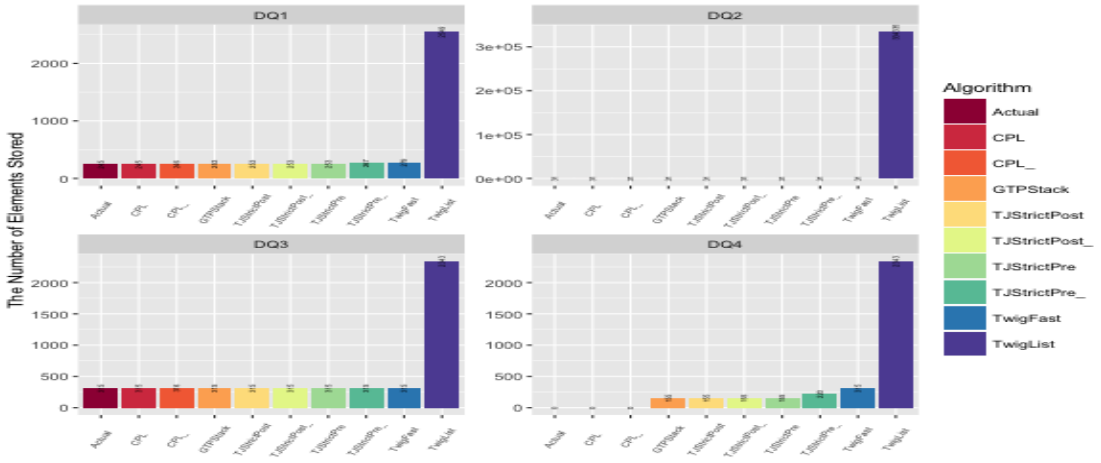


Figure C.1: The number of elements stored in the intermediate storage by each algorithm for the queries tested over DBLP. Actual represents the number of elements relevant to the query results.

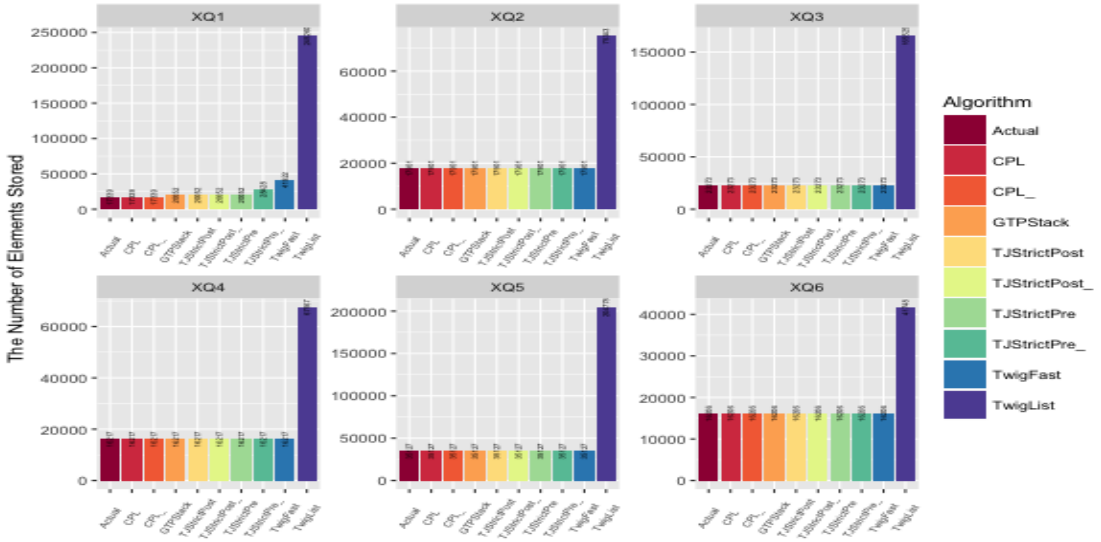


Figure C.2: The number of elements stored in the intermediate storage by each algorithm for the queries tested over XMark. Actual represents the number of elements relevant to the query results.

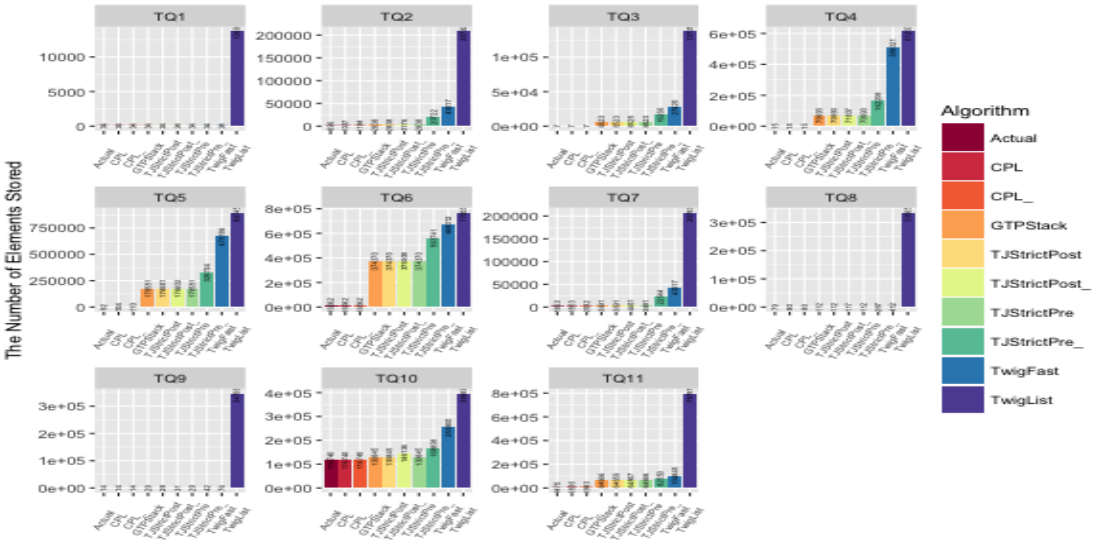


Figure C.3: The number of elements stored in the intermediate storage by each algorithm for the queries tested over the TreeBank document. Actual represents the number of elements relevant to the query results.

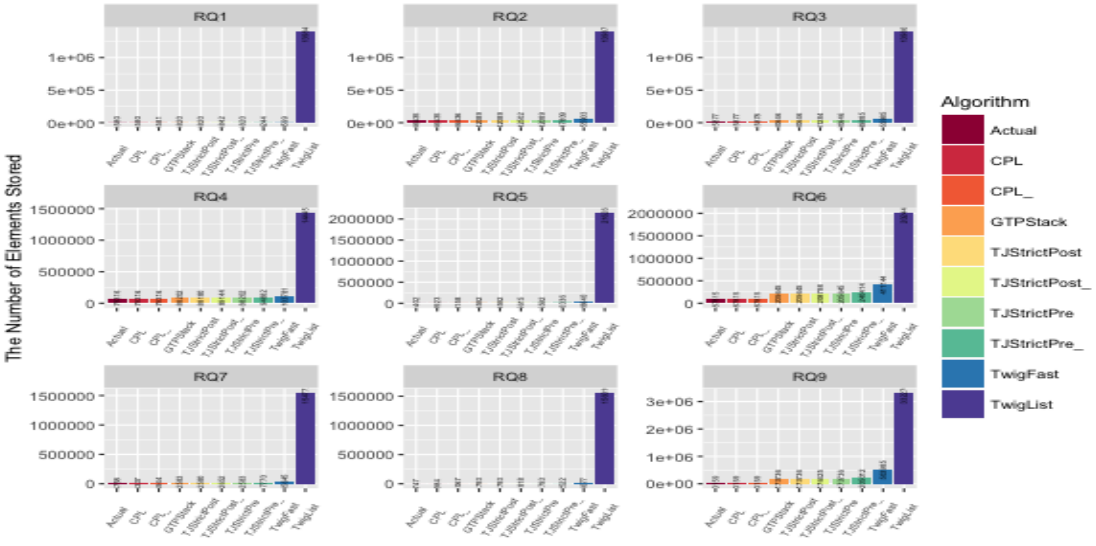


Figure C.4: The number of elements stored in the intermediate storage by each algorithm for the queries tested over the Random dataset. Actual represents the number of elements relevant to the query results.

Table C.2: Processing times for the DBLP dataset.

Algorithm	DQ1	DQ2	DQ3	DQ4
TwigList	1.461	0.853	0.964	0.965
TwigFast	0.288	0.955	0.1885	0.189
TwigPrime_	0.385	0.988	0.188	0.301
TwigPrime_N	0.382	0.992	0.188	0.301
TwigPrime	0.3895	1.064	0.189	0.3
TwigPrimePart_	0.36	0.001	0.196	0.101
TwigPrimePart_N	0.36	0.001	0.196	0.101
TwigPrimeMatch_	0.3605	0.024	0.1925	0.112
TwigPrimeMatch_N	0.362	0.009	0.194	0.112
TwigPrimePart	0.365	0.001	0.192	0.102
TwigPrimeMatch	0.364	0.941	0.183	0.2945
TJStrictPre	0.278	0.167	0.187	0.185
TJStrictPre_	0.276	0.1685	0.19	0.187
TJStrictPrePrime	0.361	0.004	0.192	0.102
TJStrictPrePrime_	0.36	0.004	0.192	0.101
TJStrictPost	0.277	0.169	0.188	0.187
TJStrictPost_	0.273	0.169	0.1895	0.188
TJStrictPostPrime	0.361	0.004	0.194	0.1
TJStrictPostPrime_	0.361	0.004	0.191	0.1
GTPStack	0.275	0.167	0.1845	0.1825
GTPStackPrime	0.361	0.024	0.197	0.111

Table C.3: Processing times for the XMark dataset.

Algorithm	XQ1	XQ2	XQ3	XQ4	XQ5	XQ6
TwigFast	2.8605	0.214	0.507	0.511	1.2925	0.35
TwigPrime_	4.3435	0.221	0.807	0.837	1.546	0.681
TwigPrime_N	4.245	0.229	0.817	0.813	1.41	0.6165
TwigPrime	4.4735	0.2705	0.918	0.926	1.643	0.734
TwigPrimePart_	2.829	0.2055	0.625	1.101	1.5435	1.241
TwigPrimePart_N	2.8255	0.205	0.6265	1.0965	1.5345	1.2435
TwigPrimeMatch_	0.8575	0.193	0.458	0.618	0.872	0.7955
TwigPrimeMatch_N	0.852	0.192	0.459	0.6195	0.873	0.7975
TwigPrimePart	2.321	0.209	0.562	0.923	1.4455	1.059
TwigPrimeMatch	2.864	0.281	0.9645	1.114	1.229	0.832
TJStrictPre	1.791	0.1945	0.382	0.01	1.253	0.6425
TJStrictPre_	1.6635	0.167	0.315	0.01	1.168	0.5945
TJStrictPrePrime	0.3745	0.199	0.508	0.788	1.42	1.065
TJStrictPrePrime_	0.361	0.1685	0.435	0.7225	1.386	1.028
TJStrictPost	1.6685	0.215	0.405	0.01	1.1305	0.579
TJStrictPost_	1.595	0.186	0.343	0.01	1.1035	0.557
TJStrictPostPrime	0.3675	0.215	0.5265	0.749	1.299	0.986
TJStrictPostPrime_	0.36	0.1865	0.4635	0.699	1.268	0.968
GTPStack	0.75	1.799	2.051	0.445	7.9795	0.579
GTPStackPrime	1.0075	1.8035	2.143	0.743	12.9965	0.9585

Table C.4: Processing times for the TreeBank dataset.

Algorithm	TQ1	TQ10	TQ11	TQ2	TQ3	TQ4	TQ5	TQ6	TQ7	TQ8	TQ9
TwigList	0.361	2.6855	25.0335	4.019	0.6995	4.131	16.9675	9.645	3.922	5.245	3.363
TwigFast	0.103	2.0425	32.4555	1.818	0.444	4.0655	18.111	10.8295	1.8105	2.349	2.6
TwigPrime_	0.103	2.1815	22.23	2.0255	0.6105	1.961	5.3485	3.522	2.0315	3.62	2.7545
TwigPrime_N	0.104	2.0455	22.1775	2.006	0.609	1.9545	5.336	3.4275	2.022	2.94	2.739
TwigPrime	0.105	2.344	22.586	2.08	0.6405	2.092	5.642	3.6965	2.0955	3.835	2.8485
TwigPrimePart_	0.069	2.0715	2.023	1.1385	0.173	0.326	0.612	0.846	1.1635	0.778	0.591
TwigPrimePart_N	0.069	2.074	2.0185	1.139	0.173	0.3275	0.614	0.845	1.1745	0.667	0.595
TwigPrimeMatch_	0.073	1.988	1.402	0.9665	0.174	0.325	0.6285	0.7835	0.98	0.78	0.5835
TwigPrimeMatch_N	0.073	1.9905	1.406	0.969	0.173	0.327	0.632	0.789	0.982	0.67	0.588
TwigPrimePart	0.068	2.0985	1.865	1.09	0.172	0.3255	0.607	0.81	1.114	0.763	0.586
TwigPrimeMatch	0.103	2.4945	11.6445	2.4945	0.638	2.081	6.8545	5.109	2.524	2.391	2.6525
TJStrictPre	0.071	2.1255	5.222	1.7325	0.2965	2.914	9.4795	7.3905	1.75	0.587	0.409
TJStrictPre_	0.072	2.1505	5.3055	1.731	0.2975	2.8975	8.562	7.6025	1.7475	0.594	0.414
TJStrictPrePrime	0.073	1.9115	1.8445	1.076	0.171	0.327	0.6045	0.799	1.091	0.653	0.584
TJStrictPrePrime_	0.073	1.852	1.833	1.06	0.169	0.3265	0.6035	0.7835	1.081	0.653	0.582
TJStrictPost	0.072	2.314	5.254	1.72	0.316	3.15	8.784	7.6425	1.7315	0.592	0.4135
TJStrictPost_	0.072	2.3625	5.275	1.726	0.317	3.1335	8.7715	7.696	1.735	0.712	0.414
TJStrictPostPrime	0.073	2.0255	1.8345	1.064	0.17	0.3245	0.603	0.799	1.076	0.654	0.581
TJStrictPostPrime_	0.072	1.997	1.838	1.068	0.171	0.323	0.5975	0.786	1.085	0.652	0.58
GTPStack	0.071	184.953	4.059	0.8555	0.3215	4.025	7.178	291.776	0.866	0.6295	0.41
GTPStackPrime	0.0735	193.416	1.487	0.989	0.1745	0.327	0.6215	3.9175	1.002	0.8065	0.591

Table C.5: Processing times for the Random dataset.

Algorithm	RQ1	RQ2	RQ3	RQ4	RQ5	RQ6	RQ7	RQ8	RQ9
TwigList	10.2905	7.792	8.1685	8.194	15.364	13.1315	11.247	14.8805	78.479
TwigFast	4.494	4.4345	4.435	5.9135	9.089	10.9585	6.1205	9.2805	68.5225
TwigPrime_	5.3615	5.281	4.8985	7.5145	11.9025	7.828	6.729	12.9015	15.008
TwigPrime_N	5.406	5.09	4.7915	6.996	11.9305	7.6285	6.803	12.995	15.0005
TwigPrime	5.8135	5.718	5.306	7.9785	12.6145	8.4475	7.436	13.562	16.004
TwigPrimePart_	3.823	4.094	3.6435	5.1125	4.954	4.3155	4.3815	4.954	4.5795
TwigPrimePart_N	3.822	4.093	3.646	5.138	4.948	4.285	4.4025	4.932	4.5825
TwigPrimeMatch_	3.1575	3.2675	2.8215	4.266	4.011	3.92	3.5615	4.173	4.0655
TwigPrimeMatch_N	3.165	3.245	2.82	4.247	3.954	3.81	3.5865	4.177	3.812
TwigPrimePart	3.472	3.7045	3.3145	4.642	4.5325	4.079	4.018	4.5275	4.2765
TwigPrimeMatch	5.882	5.9065	5.404	10.2465	9.023	8.572	6.6385	7.395	16.183
TJStrictPre	2.905	3.2195	3.164	3.9505	4.199	5.1975	3.83	3.586	8.0415
TJStrictPre_	2.968	3.2315	3.18	3.9995	4.2195	5.1925	3.8155	3.601	8.061
TJStrictPrePrime	3.3935	3.6255	3.2605	4.5665	4.4815	3.9355	3.9535	4.4515	4.203
TJStrictPrePrime_	3.416	3.6165	3.252	4.5285	4.457	3.8975	3.9575	4.453	4.185
TJStrictPost	2.9595	3.116	3.097	3.7005	4.1955	5.487	3.786	3.5875	8.369
TJStrictPost_	2.9265	3.111	3.079	3.6775	4.195	5.559	3.791	3.59	8.4165
TJStrictPostPrime	3.3885	3.546	3.233	4.3295	4.462	4.044	3.9355	4.4445	4.211
TJStrictPostPrime_	3.3935	3.5275	3.1655	4.289	4.438	3.976	3.904	4.4265	4.1695
GTPStack	2.4715	11.022	4.8805	110.9105	3.1455	239.249	2.8895	2.71	17.414
GTPStackPrime	3.132	10.001	4.918	98.831	3.7835	51.9045	3.492	3.9725	3.571

Table C.6: Processing times for the Zipf dataset, Template T_1.

Algorithm	ZQ1	ZQ2	ZQ3	ZQ4	ZQ5	ZQ6	ZQ7	ZQ8	ZQ9	ZQ10
TwigPrime	12.69	6.331	6.3285	10.182	4.985	8.383	8.1065	4.972	8.803	9.677
TwigPrimePart	10.769	5.9185	4.2475	3.5625	6.087	3.9145	6.118	5.65	3.581	6.189
TwigPrimeMatch	13.4005	6.4785	6.578	10.171	5.144	8.436	8.161	5.14	8.8995	10.0165
TJStrictPre	12.174	7.497	6.687	5.774	6.884	7.5785	8.139	6.8705	7.533	9.823
TJStrictPrePrime	10.4945	5.6685	4.168	3.407	5.75	3.7535	6.017	5.355	3.4425	5.7285
GTPStack	368.49	370.869	1157.271	179.237	236.982	321.85	84.643	360.021	863.928	1371.898
GTPStackPrime	182.265	41.1545	175.891	76.179	36.9995	16.0905	51.9345	42.306	77.853	411.799

Table C.7: Processing times for the Zipf dataset, Template T_2.

Algorithm	ZQ11	ZQ12	ZQ13	ZQ14	ZQ15	ZQ16	ZQ17	ZQ18	ZQ19	ZQ20
TwigPrimePart	0.9095	2.961	5.1895	1.2565	1.1585	3.469	1.957	6.738	2.4715	1.6995
TwigPrimeMatch	2.389	3.7455	6.1195	3.6345	1.8965	7.276	2.874	9.554	3.102	5.639
TJStrictPre	3.155	5.692	8.3985	4.692	2.692	8.4245	4.1895	10.7375	4.647	7.0055
TJStrictPrePrime	0.871	2.757	4.675	1.1995	1.131	3.0575	1.829	5.793	2.372	1.5875
GTPStack	4.537	14.455	109.325	7.0515	3.746	18.9145	6.486	92.2085	8.65	10.944
GTPStackPrime	0.7965	10.9615	97.4935	1.481	1.0155	11.065	2.599	86.261	5.1595	2.366

Table C.8: Processing times for the Zipf dataset, Template T_3.

Algorithm	ZQ21	ZQ22	ZQ23	ZQ24	ZQ25	ZQ26	ZQ27	ZQ28	ZQ29	ZQ30
TwigPrimeMatch	3.844	10.9675	7.6135	9.588	9.187	4.7595	6.529	3.3635	4.4845	5.6555
TJStrictPre	5.6415	14.5685	11.604	14.3135	15.3165	7.8035	11.21	4.8535	7.216	7.364
TJStrictPrePrime	1.104	3.9825	5.162	5.244	4.201	2.479	3.369	1.0265	2.992	1.963
GTPStack	64.101	1582.083	603.838	449.324	3553.837	187.41	1128.867	17.072	395.912	59.5225
GTPStackPrime	1.005	138.281	135.884	85.804	522.379	6.635	40.759	0.889	22.5275	3.0165

Table C.9: Processing times for the Zipf dataset, Template T_4.

Algorithm	ZQ31	ZQ32	ZQ33	ZQ34	ZQ35	ZQ36	ZQ37	ZQ38	ZQ39	ZQ40
TwigPrime	3.234	5.236	5.6915	4.631	7.18	3.4865	3.167	6.2855	6.136	7.4585
TwigPrimePart	4.381	5.37	4.202	4.8745	4.204	4.269	4.053	4.258	4.3875	4.993
TwigPrimeMatch	3.1105	5.168	6.5385	4.9175	7.9635	3.697	3.2275	6.3905	6.3545	8.187
TJStrictPre	4.393	5.927	4.8025	4.869	4.802	4.681	4.127	4.405	4.729	6.697
TJStrictPrePrime	4.239	5.1145	4.054	4.619	4.021	4.142	3.9225	4.0745	4.169	4.769
GTPStack	14.9635	20.8745	12.774	32.467	11.2315	18.2175	13.3595	9.33	14.804	17.907
GTPStackPrime	13.413	20.491	13.461	31.382	11.555	18.5365	12.641	9.4935	13.4925	18.707

Table C.10: Processing times for the Zipf dataset, Template T_5.

Algorithm	ZQ41	ZQ42	ZQ43	ZQ44	ZQ45	ZQ46	ZQ47	ZQ48	ZQ49	ZQ50
TwigPrime	15.0545	12.169	25.781	27.465	18.3795	15.839	10.346	15.965	11.8455	27.1075
TwigPrimePart	3.7495	5.633	5.906	2.7325	2.581	4.457	5.4675	4.56	3.807	3.821
TwigPrimeMatch	13.4975	10.441	16.17	17.9015	13.6715	10.972	7.9825	11.1455	10.804	17.6265

Table C.10: Processing times for the Zipf dataset, Template T_5.

Algorithm	ZQ41	ZQ42	ZQ43	ZQ44	ZQ45	ZQ46	ZQ47	ZQ48	ZQ49	ZQ50
TJStrictPre	9.833	14.0605	29.229	9.6225	5.998	10.241	16.1525	10.5045	6.972	14.8075
TJStrictPrePrime	3.685	5.634	5.891	2.712	2.583	4.416	5.412	4.5285	3.7935	3.7765
GTPStack	4.7655	5.336	4.1215	3.4415	2.7505	4.236	5.0065	4.349	3.412	4.7215
GTPStackPrime	3.6535	5.055	3.179	2.5555	2.518	4.67	4.255	3.897	3.743	3.1755

Bottom Up Results for Ordered Axes, Sequence Operators and Positional Predicates

Table C.11: Results for paired comparisons based on the U test over the XMark dataset, Experiment 1.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	medina A	Median B	effect size r	Best
OTwigPrimeList	OPTwigPrime	OXQ1	2.50E-34	TRUE	1.046	0.7945	0.8599292	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OXQ2	7.51E-31	TRUE	1.0785	0.5165	0.812377	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OXQ3	2.47E-32	TRUE	0.492	0.622	0.8329845	OTwigPrimeList
OTwigPrimeList	OPTwigPrime	OXQ4	2.48E-34	TRUE	1.2745	0.924	0.8599647	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OXQ5	2.53E-34	TRUE	1.1945	0.848	0.8598687	OPTwigPrime
OTwigPrimeList	TwigPos	OXQ1	2.53E-34	TRUE	1.046	1.226	0.8598693	OTwigPrimeList
OTwigPrimeList	TwigPos	OXQ2	2.59E-26	TRUE	1.0785	1.5735	0.7458713	OTwigPrimeList
OTwigPrimeList	TwigPos	OXQ3	7.11E-34	TRUE	0.492	0.6865	0.8538723	OTwigPrimeList
OTwigPrimeList	TwigPos	OXQ4	2.53E-34	TRUE	1.2745	1.6455	0.8598505	OTwigPrimeList
OTwigPrimeList	TwigPos	OXQ5	4.62E-13	TRUE	1.1945	1.172	0.504974	TwigPos
OPTwigPrime	TwigPos	OXQ1	2.51E-34	TRUE	0.7945	1.226	0.8598951	OPTwigPrime
OPTwigPrime	TwigPos	OXQ2	4.68E-33	TRUE	0.5165	1.5735	0.8428446	OPTwigPrime
OPTwigPrime	TwigPos	OXQ3	4.62E-33	TRUE	0.622	0.6865	0.842918	OPTwigPrime
OPTwigPrime	TwigPos	OXQ4	2.49E-34	TRUE	0.924	1.6455	0.8599436	OPTwigPrime
OPTwigPrime	TwigPos	OXQ5	2.53E-34	TRUE	0.848	1.172	0.8598609	OPTwigPrime

Table C.12: Results for paired comparisons based on the U test over the TreeBank dataset, Experiment 1.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	medina A	Median B	effect size r	Best
OTwigPrimeList	OPTwigPrime	OTQ1	2.46E-34	TRUE	5.2565	2.129	0.8600227	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ2	2.46E-34	TRUE	3.5935	2.527	0.860022	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ3	2.44E-34	TRUE	0.745	0.201	0.8600761	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ4	7.67E-19	TRUE	1.9835	1.6975	0.6213435	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ5	2.45E-34	TRUE	2.3845	0.837	0.8600442	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ6	2.42E-34	TRUE	2.1305	0.544	0.860126	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ7	2.46E-34	TRUE	2.7605	0.9735	0.860012	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ8	2.45E-34	TRUE	1.469	1.203	0.8600536	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ9	1.83E-34	TRUE	0.106	0.075	0.8617208	OPTwigPrime
OTwigPrimeList	TwigPos	OTQ1	2.46E-34	TRUE	5.2565	6.903	0.860011	OTwigPrimeList
OTwigPrimeList	TwigPos	OTQ2	5.05E-01	FALSE	3.5935	3.5345	0.00096002	tie
OTwigPrimeList	TwigPos	OTQ3	8.71E-33	TRUE	0.745	1.08	0.8391729	OTwigPrimeList
OTwigPrimeList	TwigPos	OTQ4	2.46E-34	TRUE	1.9835	3.009	0.8600282	OTwigPrimeList
OTwigPrimeList	TwigPos	OTQ5	2.46E-34	TRUE	2.3845	3.196	0.8600256	OTwigPrimeList
OTwigPrimeList	TwigPos	OTQ6	2.46E-34	TRUE	2.1305	2.811	0.8600318	OTwigPrimeList
OTwigPrimeList	TwigPos	OTQ7	2.47E-34	TRUE	2.7605	4.249	0.8600051	OTwigPrimeList
OTwigPrimeList	TwigPos	OTQ8	2.45E-34	TRUE	1.469	3.6305	0.8600377	OTwigPrimeList
OTwigPrimeList	TwigPos	OTQ9	2.11E-34	TRUE	0.106	0.475	0.8608981	OTwigPrimeList
OPTwigPrime	TwigPos	OTQ1	2.46E-34	TRUE	2.129	6.903	0.8600194	OPTwigPrime
OPTwigPrime	TwigPos	OTQ2	2.46E-34	TRUE	2.527	3.5345	0.860023	OPTwigPrime
OPTwigPrime	TwigPos	OTQ3	2.44E-34	TRUE	0.201	1.08	0.8600761	OPTwigPrime
OPTwigPrime	TwigPos	OTQ4	2.45E-34	TRUE	1.6975	3.009	0.8600442	OPTwigPrime
OPTwigPrime	TwigPos	OTQ5	2.44E-34	TRUE	0.837	3.196	0.8600602	OPTwigPrime
OPTwigPrime	TwigPos	OTQ6	2.42E-34	TRUE	0.544	2.811	0.8601181	OPTwigPrime
OPTwigPrime	TwigPos	OTQ7	2.46E-34	TRUE	0.9735	4.249	0.860012	OPTwigPrime
OPTwigPrime	TwigPos	OTQ8	2.44E-34	TRUE	1.203	3.6305	0.8600608	OPTwigPrime
OPTwigPrime	TwigPos	OTQ9	2.06E-34	TRUE	0.075	0.475	0.8610409	OPTwigPrime

Table C.13: Results for paired comparisons based on the U test over the Random dataset, Experiment 1.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	medina A	Median B	effect size r	Best
OTwigPrimeList	OPTwigPrime	ORQ1	2.55E-34	TRUE	10.912	4.1785	0.8598043	OPTwigPrime
OTwigPrimeList	OPTwigPrime	ORQ2	3.52E-33	TRUE	10.914	4.7985	0.8445152	OPTwigPrime
OTwigPrimeList	OPTwigPrime	ORQ3	2.55E-34	TRUE	6.1455	3.2425	0.8598072	OPTwigPrime
OTwigPrimeList	OPTwigPrime	ORQ4	1.16E-23	TRUE	3.891	2.822	0.7041548	OPTwigPrime
OTwigPrimeList	OPTwigPrime	ORQ5	2.55E-34	TRUE	8.6225	4.0455	0.8598075	OPTwigPrime
OTwigPrimeList	OPTwigPrime	ORQ6	2.55E-34	TRUE	17.579	3.516	0.8598124	OPTwigPrime
OTwigPrimeList	OPTwigPrime	ORQ7	2.55E-34	TRUE	6.6695	3.6355	0.8598121	OPTwigPrime
OTwigPrimeList	TwigPos	ORQ1	2.56E-34	TRUE	10.912	16.4675	0.8597929	OTwigPrimeList
OTwigPrimeList	TwigPos	ORQ2	0.000926213	TRUE	10.914	11.1975	0.2201171	OTwigPrimeList
OTwigPrimeList	TwigPos	ORQ3	2.56E-34	TRUE	6.1455	14.098	0.8597961	OTwigPrimeList
OTwigPrimeList	TwigPos	ORQ4	2.20E-34	TRUE	3.891	136.962	0.8606767	OTwigPrimeList
OTwigPrimeList	TwigPos	ORQ5	2.56E-34	TRUE	8.6225	18.0805	0.8597929	OTwigPrimeList
OTwigPrimeList	TwigPos	ORQ6	2.56E-34	TRUE	17.579	24.965	0.8597938	OTwigPrimeList
OTwigPrimeList	TwigPos	ORQ7	2.56E-34	TRUE	6.6695	13.944	0.8597994	OTwigPrimeList
OPTwigPrime	TwigPos	ORQ1	2.56E-34	TRUE	4.1785	16.4675	0.859799	OPTwigPrime
OPTwigPrime	TwigPos	ORQ2	2.33E-33	TRUE	4.7985	11.1975	0.8469442	OPTwigPrime
OPTwigPrime	TwigPos	ORQ3	2.56E-34	TRUE	3.2425	14.098	0.8598013	OPTwigPrime
OPTwigPrime	TwigPos	ORQ4	2.20E-34	TRUE	2.822	136.962	0.8606767	OPTwigPrime
OPTwigPrime	TwigPos	ORQ5	2.56E-34	TRUE	4.0455	18.0805	0.8598023	OPTwigPrime
OPTwigPrime	TwigPos	ORQ6	2.55E-34	TRUE	3.516	24.965	0.8598075	OPTwigPrime
OPTwigPrime	TwigPos	ORQ7	2.56E-34	TRUE	3.6355	13.944	0.8598016	OPTwigPrime

Table C.14: Results for paired comparisons based on the U test over the XMark dataset, Experiment 2.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	medina A	Median B	effect size r	Best
OPTwigPrime	TwigPos	PXQ1	2.52E-34	TRUE	2.8325	0.424	0.8598752	TwigPos
OPTwigPrime	TwigPos	PXQ2	4.84E-33	TRUE	1.614	1.92	0.842648	OPTwigPrime
OPTwigPrime	TwigPos	PXQ3	7.70E-32	TRUE	0.969	0.8285	0.8261779	TwigPos
OPTwigPrime	TwigPos	PXQ4	2.52E-34	TRUE	0.744	1.171	0.8598856	OPTwigPrime
OPTwigPrime	TwigPos	PXQ5	1.41E-15	TRUE	1.055	1.016	0.5585102	TwigPos
OPTwigPrime	TwigPos	PXQ6	1.58E-05	TRUE	1.748	1.771	0.2943182	OPTwigPrime
OPTwigPrime	TwigPos	PXQ7	1.07E-05	TRUE	2.278	2.3065	0.3005597	OPTwigPrime
OPTwigPrime	TwigPos	PXQ8	2.54E-34	TRUE	0.528	18.927	0.8598485	OPTwigPrime

Table C.15: Results for paired comparisons based on the U test over the TreeBank dataset, Experiment 2.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	medina A	Median B	effect size r	Best
OPTwigPrime	TwigPos	PTQ1	2.55E-34	TRUE	1.4805	6.6695	0.8598033	OPTwigPrime
OPTwigPrime	TwigPos	PTQ2	2.53E-34	TRUE	0.605	2.3675	0.8598661	OPTwigPrime
OPTwigPrime	TwigPos	PTQ3	2.55E-34	TRUE	2.2185	4.029	0.8598033	OPTwigPrime
OPTwigPrime	TwigPos	PTQ4	1.75E-24	TRUE	6.729	5.921	0.7173423	TwigPos
OPTwigPrime	TwigPos	PTQ5	2.47E-34	TRUE	0.269	1.973	0.8600028	OPTwigPrime
OPTwigPrime	TwigPos	PTQ6	2.55E-34	TRUE	1.112	6.3875	0.8598205	OPTwigPrime
OPTwigPrime	TwigPos	PTQ7	2.55E-34	TRUE	1.138	5.6025	0.8598235	OPTwigPrime
OPTwigPrime	TwigPos	PTQ8	2.55E-34	TRUE	1.852	3.3035	0.859814	OPTwigPrime

Table C.16: Results for paired comparisons based on the U test over the Random dataset, Experiment 2.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	medina A	Median B	effect size r	Best
OPTwigPrime	TwigPos	RQ1	2.56E-34	TRUE	5.1875	16.519	0.859789	OPTwigPrime
OPTwigPrime	TwigPos	RQ2	4.95E-34	TRUE	4.311	12.2165	0.8559711	OPTwigPrime
OPTwigPrime	TwigPos	RQ3	2.56E-34	TRUE	4.8725	21.8965	0.8597896	OPTwigPrime
OPTwigPrime	TwigPos	RQ4	2.56E-34	TRUE	5.303	12.4455	0.8597899	OPTwigPrime
OPTwigPrime	TwigPos	RQ5	2.56E-34	TRUE	5.4295	21.1635	0.8597903	OPTwigPrime
OPTwigPrime	TwigPos	RQ6	2.56E-34	TRUE	3.997	16.386	0.8597909	OPTwigPrime
OPTwigPrime	TwigPos	RQ7	2.56E-34	TRUE	4.2495	15.976	0.8597916	OPTwigPrime
OPTwigPrime	TwigPos	RQ8	2.56E-34	TRUE	5.7485	23.16	0.8597899	OPTwigPrime

Appendix D

Ordered and Positional Bottom-Up Holistic Approaches Full results

Table D.1: Results for paired comparisons based on the U test over the XMark dataset, Experiment 1.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	medina A	Median B	effect size r	Best
OTwigPrimeList	OPTwigPrime	OXQ1	2.50E-34	TRUE	1.046	0.7945	0.8599292	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OXQ2	7.51E-31	TRUE	1.0785	0.5165	0.812377	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OXQ3	2.47E-32	TRUE	0.492	0.622	0.8329845	OTwigPrimeList
OTwigPrimeList	OPTwigPrime	OXQ4	2.48E-34	TRUE	1.2745	0.924	0.8599647	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OXQ5	2.53E-34	TRUE	1.1945	0.848	0.8598687	OPTwigPrime
OTwigPrimeList	TwigPos	OXQ1	2.53E-34	TRUE	1.046	1.226	0.8598693	OTwigPrimeList
OTwigPrimeList	TwigPos	OXQ2	2.59E-26	TRUE	1.0785	1.5735	0.7458713	OTwigPrimeList
OTwigPrimeList	TwigPos	OXQ3	7.11E-34	TRUE	0.492	0.6865	0.8538723	OTwigPrimeList
OTwigPrimeList	TwigPos	OXQ4	2.53E-34	TRUE	1.2745	1.6455	0.8598505	OTwigPrimeList
OTwigPrimeList	TwigPos	OXQ5	4.62E-13	TRUE	1.1945	1.172	0.504974	TwigPos
OPTwigPrime	TwigPos	OXQ1	2.51E-34	TRUE	0.7945	1.226	0.8598951	OPTwigPrime
OPTwigPrime	TwigPos	OXQ2	4.68E-33	TRUE	0.5165	1.5735	0.8428446	OPTwigPrime
OPTwigPrime	TwigPos	OXQ3	4.62E-33	TRUE	0.622	0.6865	0.842918	OPTwigPrime
OPTwigPrime	TwigPos	OXQ4	2.49E-34	TRUE	0.924	1.6455	0.8599436	OPTwigPrime
OPTwigPrime	TwigPos	OXQ5	2.53E-34	TRUE	0.848	1.172	0.8598609	OPTwigPrime

Table D.2: Results for paired comparisons based on the U test over the TreeBank dataset, Experiment 1.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	medina A	Median B	effect size r	Best
OTwigPrimeList	OPTwigPrime	OTQ1	2.46E-34	TRUE	5.2565	2.129	0.8600227	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ2	2.46E-34	TRUE	3.5935	2.527	0.860022	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ3	2.44E-34	TRUE	0.745	0.201	0.8600761	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ4	7.67E-19	TRUE	1.9835	1.6975	0.6213435	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ5	2.45E-34	TRUE	2.3845	0.837	0.8600442	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ6	2.42E-34	TRUE	2.1305	0.544	0.860126	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ7	2.46E-34	TRUE	2.7605	0.9735	0.860012	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ8	2.45E-34	TRUE	1.469	1.203	0.8600536	OPTwigPrime
OTwigPrimeList	OPTwigPrime	OTQ9	1.83E-34	TRUE	0.106	0.075	0.8617208	OPTwigPrime
OTwigPrimeList	TwigPos	OTQ1	2.46E-34	TRUE	5.2565	6.903	0.860011	OTwigPrimeList
OTwigPrimeList	TwigPos	OTQ2	5.05E-01	FALSE	3.5935	3.5345	0.00096002	tie
OTwigPrimeList	TwigPos	OTQ3	8.71E-33	TRUE	0.745	1.08	0.8391729	OTwigPrimeList
OTwigPrimeList	TwigPos	OTQ4	2.46E-34	TRUE	1.9835	3.009	0.8600282	OTwigPrimeList
OTwigPrimeList	TwigPos	OTQ5	2.46E-34	TRUE	2.3845	3.196	0.8600256	OTwigPrimeList
OTwigPrimeList	TwigPos	OTQ6	2.46E-34	TRUE	2.1305	2.811	0.8600318	OTwigPrimeList
OTwigPrimeList	TwigPos	OTQ7	2.47E-34	TRUE	2.7605	4.249	0.8600051	OTwigPrimeList
OTwigPrimeList	TwigPos	OTQ8	2.45E-34	TRUE	1.469	3.6305	0.8600377	OTwigPrimeList
OTwigPrimeList	TwigPos	OTQ9	2.11E-34	TRUE	0.106	0.475	0.8608981	OTwigPrimeList
OPTwigPrime	TwigPos	OTQ1	2.46E-34	TRUE	2.129	6.903	0.8600194	OPTwigPrime
OPTwigPrime	TwigPos	OTQ2	2.46E-34	TRUE	2.527	3.5345	0.860023	OPTwigPrime
OPTwigPrime	TwigPos	OTQ3	2.44E-34	TRUE	0.201	1.08	0.8600761	OPTwigPrime
OPTwigPrime	TwigPos	OTQ4	2.45E-34	TRUE	1.6975	3.009	0.8600442	OPTwigPrime
OPTwigPrime	TwigPos	OTQ5	2.44E-34	TRUE	0.837	3.196	0.8600602	OPTwigPrime
OPTwigPrime	TwigPos	OTQ6	2.42E-34	TRUE	0.544	2.811	0.8601181	OPTwigPrime
OPTwigPrime	TwigPos	OTQ7	2.46E-34	TRUE	0.9735	4.249	0.860012	OPTwigPrime
OPTwigPrime	TwigPos	OTQ8	2.44E-34	TRUE	1.203	3.6305	0.8600608	OPTwigPrime
OPTwigPrime	TwigPos	OTQ9	2.06E-34	TRUE	0.075	0.475	0.8610409	OPTwigPrime

Table D.3: Results for paired comparisons based on the U test over the Random dataset, Experiment 1.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	medina A	Median B	effect size r	Best
OTwigPrimeList	OPTwigPrime	ORQ1	2.55E-34	TRUE	10.912	4.1785	0.8598043	OPTwigPrime
OTwigPrimeList	OPTwigPrime	ORQ2	3.52E-33	TRUE	10.914	4.7985	0.8445152	OPTwigPrime
OTwigPrimeList	OPTwigPrime	ORQ3	2.55E-34	TRUE	6.1455	3.2425	0.8598072	OPTwigPrime
OTwigPrimeList	OPTwigPrime	ORQ4	1.16E-23	TRUE	3.891	2.822	0.7041548	OPTwigPrime
OTwigPrimeList	OPTwigPrime	ORQ5	2.55E-34	TRUE	8.6225	4.0455	0.8598075	OPTwigPrime
OTwigPrimeList	OPTwigPrime	ORQ6	2.55E-34	TRUE	17.579	3.516	0.8598124	OPTwigPrime
OTwigPrimeList	OPTwigPrime	ORQ7	2.55E-34	TRUE	6.6695	3.6355	0.8598121	OPTwigPrime
OTwigPrimeList	TwigPos	ORQ1	2.56E-34	TRUE	10.912	16.4675	0.8597929	OTwigPrimeList
OTwigPrimeList	TwigPos	ORQ2	0.000926213	TRUE	10.914	11.1975	0.2201171	OTwigPrimeList
OTwigPrimeList	TwigPos	ORQ3	2.56E-34	TRUE	6.1455	14.098	0.8597961	OTwigPrimeList
OTwigPrimeList	TwigPos	ORQ4	2.20E-34	TRUE	3.891	136.962	0.8606767	OTwigPrimeList
OTwigPrimeList	TwigPos	ORQ5	2.56E-34	TRUE	8.6225	18.0805	0.8597929	OTwigPrimeList
OTwigPrimeList	TwigPos	ORQ6	2.56E-34	TRUE	17.579	24.965	0.8597938	OTwigPrimeList
OTwigPrimeList	TwigPos	ORQ7	2.56E-34	TRUE	6.6695	13.944	0.8597994	OTwigPrimeList
OPTwigPrime	TwigPos	ORQ1	2.56E-34	TRUE	4.1785	16.4675	0.859799	OPTwigPrime
OPTwigPrime	TwigPos	ORQ2	2.33E-33	TRUE	4.7985	11.1975	0.8469442	OPTwigPrime
OPTwigPrime	TwigPos	ORQ3	2.56E-34	TRUE	3.2425	14.098	0.8598013	OPTwigPrime
OPTwigPrime	TwigPos	ORQ4	2.20E-34	TRUE	2.822	136.962	0.8606767	OPTwigPrime
OPTwigPrime	TwigPos	ORQ5	2.56E-34	TRUE	4.0455	18.0805	0.8598023	OPTwigPrime
OPTwigPrime	TwigPos	ORQ6	2.55E-34	TRUE	3.516	24.965	0.8598075	OPTwigPrime
OPTwigPrime	TwigPos	ORQ7	2.56E-34	TRUE	3.6355	13.944	0.8598016	OPTwigPrime

Table D.4: Results for paired comparisons based on the U test over the XMark dataset, Experiment 2.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	medina A	Median B	effect size r	Best
OPTwigPrime	TwigPos	PXQ1	2.52E-34	TRUE	2.8325	0.424	0.8598752	TwigPos
OPTwigPrime	TwigPos	PXQ2	4.84E-33	TRUE	1.614	1.92	0.842648	OPTwigPrime
OPTwigPrime	TwigPos	PXQ3	7.70E-32	TRUE	0.969	0.8285	0.8261779	TwigPos
OPTwigPrime	TwigPos	PXQ4	2.52E-34	TRUE	0.744	1.171	0.8598856	OPTwigPrime
OPTwigPrime	TwigPos	PXQ5	1.41E-15	TRUE	1.055	1.016	0.5585102	TwigPos
OPTwigPrime	TwigPos	PXQ6	1.58E-05	TRUE	1.748	1.771	0.2943182	OPTwigPrime
OPTwigPrime	TwigPos	PXQ7	1.07E-05	TRUE	2.278	2.3065	0.3005597	OPTwigPrime
OPTwigPrime	TwigPos	PXQ8	2.54E-34	TRUE	0.528	18.927	0.8598485	OPTwigPrime

Table D.5: Results for paired comparisons based on the U test over the TreeBank dataset, Experiment 2.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	medina A	Median B	effect size r	Best
OPTwigPrime	TwigPos	PTQ1	2.55E-34	TRUE	1.4805	6.6695	0.8598033	OPTwigPrime
OPTwigPrime	TwigPos	PTQ2	2.53E-34	TRUE	0.605	2.3675	0.8598661	OPTwigPrime
OPTwigPrime	TwigPos	PTQ3	2.55E-34	TRUE	2.2185	4.029	0.8598033	OPTwigPrime
OPTwigPrime	TwigPos	PTQ4	1.75E-24	TRUE	6.729	5.921	0.7173423	TwigPos
OPTwigPrime	TwigPos	PTQ5	2.47E-34	TRUE	0.269	1.973	0.8600028	OPTwigPrime
OPTwigPrime	TwigPos	PTQ6	2.55E-34	TRUE	1.112	6.3875	0.8598205	OPTwigPrime
OPTwigPrime	TwigPos	PTQ7	2.55E-34	TRUE	1.138	5.6025	0.8598235	OPTwigPrime
OPTwigPrime	TwigPos	PTQ8	2.55E-34	TRUE	1.852	3.3035	0.859814	OPTwigPrime

Table D.6: Results for paired comparisons based on the U test over the Random dataset, Experiment 2.

AlgorithmA	AlgorithmB	Twig	P-Value	p-value < 5%	medina A	Median B	effect size r	Best
OPTwigPrime	TwigPos	RQ1	2.56E-34	TRUE	5.1875	16.519	0.859789	OPTwigPrime
OPTwigPrime	TwigPos	RQ2	4.95E-34	TRUE	4.311	12.2165	0.8559711	OPTwigPrime
OPTwigPrime	TwigPos	RQ3	2.56E-34	TRUE	4.8725	21.8965	0.8597896	OPTwigPrime
OPTwigPrime	TwigPos	RQ4	2.56E-34	TRUE	5.303	12.4455	0.8597899	OPTwigPrime
OPTwigPrime	TwigPos	RQ5	2.56E-34	TRUE	5.4295	21.1635	0.8597903	OPTwigPrime
OPTwigPrime	TwigPos	RQ6	2.56E-34	TRUE	3.997	16.386	0.8597909	OPTwigPrime
OPTwigPrime	TwigPos	RQ7	2.56E-34	TRUE	4.2495	15.976	0.8597916	OPTwigPrime
OPTwigPrime	TwigPos	RQ8	2.56E-34	TRUE	5.7485	23.16	0.8597899	OPTwigPrime