VŠB–Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Path-based Approaches to the Twig Pattern Query Searching

Ing. Radim Bača

Field of study: Computer Science and Applied Mathematics
Supervisor: Prof. RNDr. Václav Snášel, CSc.
Ostrava, 2008

Preface

XML [59] format allows a user to create annotated text in a simple way. This format can be easily read by a human as well as by a computer and the format is based on a textual representation. These features make the XML format widely popular and it can be nowadays found in many areas of usage.

A *well-formed* XML document or a set of documents can be viewed as an XML database and the associated DTD, or *XML Schema* [64], is its database schema [45]. XQuery [60] and XPath [62] are usually the query languages of an XML database.

In this thesis we study indexing approaches and query processing algorithms for a twig pattern query. The twig pattern query has been identified as a core operation of the query languages for the XML [54, 71, 5, 1, 21]. Due to this fact their performance is crucial for the overall performance of the XML query.

We study the state-of-the-art approaches for twig pattern query processing and propose their another improvement.

In Chapter 2 we make an introduction to a holistic approaches for a twig pattern query processing. Compared to other approaches they provide the best worst-case space, I/O, and time complexity in a case of optimal algorithm. We introduce some new theoretical foundations about these algorithms. Moreover, In Section 2.4 we present two novel holistic algorithms, which outperform state-of-the-art algorithms.

In Chapter 3 we describe an improvement of another existing technique called stream pruning. Some holistic algorithms use a labeled path as a main key for XML document's elements retrieval. We have to get labeled paths corresponding to a twig query pattern as a first step of a query processing. In [9] we can find definition of the stream pruning method. This technique is briefly described in Section 2.1.1 and the stream pruning algorithm is described in Section 3.1. Our work improving the original stream pruning technique can be found in Section 3.2. This work has been accepted at the IDEAS conference [B2].

In Chapter 4 we propose basic definitions for a document having an 'optimal' structure from the holistic point of view. This work has not been submitted yet, but

we consider this topic interesting since it can have a significant influence on a holistic algorithm performance.

In Chapter 5 we present our indexing method which can furthermore support the holistic algorithm. This method is mainly designed for the content-based queries. In Section 5.1 we introduce our multi-dimensional approach presented at the DEXA conference [B4]. We recognize two types of binary join algorithms, which outperform each other depending on a query. This issue is furthermore described in Section 5.2 in a context of B^+-tree indices. In Section 5.3 we introduce theoretical model for a cost-based selection of an appropriate join operation during the query processing. This algorithm has been presented at the DataX workshop [B3]. We show that our cost-based algorithm can perform significantly better then the holistic approaches in some cases.

Acknowledgment

My thank belongs to

- Prof. RNDr. Václav Snášel, CSc. for giving me the opportunity to study under his supervision,
- Doc. Michal Krátký, Ph.D. for introducing this topic to me, for many useful comments regarding databases and XML indexing, and helpful discussions from which many ideas have arisen.

I would also like to thank my girlfriend Marie for her longstanding help concerning English and for her sincere support and encouragement during my studies.

This work has been financially supported by the grants GA CR 201/06/0756 - Development of a Native Storage for XML Data. Responsible person: Jaroslav Pokorný, Charles University in Prague. Czech Science.
AV CR 1ET400300414 - Intelligent methods for increasing of reliability of electrical networks. Responsible person: Václav Snášel. Czech Science.

Contents

Chapter 1

Preliminaries

1.1 Introduction

XML (*Extensible Mark-up Language*) [59] has been recently embraced as a new approach to data modeling [45]. This mark–up language has been developed to simplify the SGML language and to create more general model for other mark–up languages. The XML format is not depended on a platform and a well formed XML document is easily readable by human or computer. XML enables simple data and or text annotations. XML Document can be very complex and there is only a few limitations on an XML document structure. This attributes of the XML technology form a powerful tool and we can find many different areas of usage today.

A *well-formed* XML document or a set of documents can be viewed as an XML database and the associated DTD, or *XML Schema* [64], is its database schema [45]. XQuery and XPath are usually the query languages of a database.

The first version of XPath [62] language was defined soon after XML. XPath language allows to specify parts of the XML document using navigation. XQuery [60] extends the XPath language and allow to specify the output more precisely. XQuery language contains many features, which makes him a very powerful language. Recently, an update facility of XQuery was proposed [61] to standardize the update language for XML.

Storage of semistructured format and an XML query processing are becoming common parts of the modern databases. The amount of data in a semistructured format is growing fast. Many applications are interchanging their data in XML and many formats are using XML as a syntax for their own XML dialect. Let us mention OpenDocument format (ODF) [43] or Office Open XML format (OOXML) [13] and

we can find hundreds of other formats in [48]. Therefore, potentially large repositories of annotated documents are created in companies and on the Internet by users who use these formats. These data can be queried by the XPath or XQuery.

If we compare the XML with the relational databases, which can also preserve structural content, there is one significant difference: the XML is a data format. That makes XML a good candidate for a long-term data storage. The format of a structured textual content is precisely defined and there is a chance that the textual content will be readable even after few decades.

As a result we need tools for XML query processing efficient even for a very large XML databases. The twig pattern query searching is often considered as a core operation of any query language for an XML data [54, 71, 5, 1, 21].

1.2 Model

An XML document can be modeled as a rooted, ordered, labeled tree, where every node of the tree corresponds to an element or an attribute of the document and the edge connect two elements or an element and attribute having the parent-child relationship. We call such representation of an XML document an *XML tree*. An example of the XML tree can be seen in Figure 1.1(b). Nodes in this XML tree are subscripted in the pre-order for easy reference. We use the term 'node' in the meaning of a node of an XML tree which represents an element or an attribute.

We introduce few terms used in a connection with the XML document D and the XML tree X. The definition are as follows:

- **Labeled path** - Let us have a node $n \in X$. *Labeled path* [49] is a sequence $tag_0/tag_1/\ldots/tag_n$ of node tags lying on the path from the root to the node n.
- **Node corresponding to the labeled path** - We say that the node n corresponds to the labeled path lp if the labeled path from the root to the node n is identical with lp.
- **Labeled path** lp **of class** tag_{lp} - By the term *labeled path* lp *of class* tag_{lp} we mean a labeled path where the tag tag_{lp} is the tag of the node $n \in X$ corresponding to the labeled path lp.
- **DataGuide** - The *DataGuide* [49] of an XML document D is an XML tree of nodes n_{DG}, where each node $n \in X$ corresponds to exactly one node n_{DG} and the labeled paths of n and n_{DG} are the same.

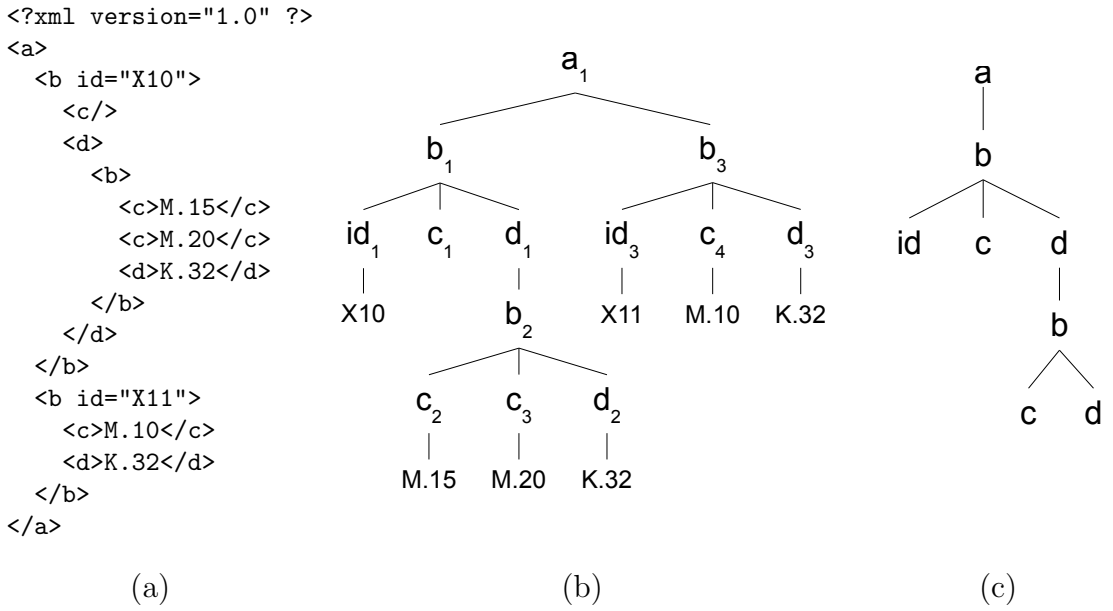


Figure 1.1: (a) XML document (b) XML tree (c) DataGuide

- TGS_D - The TGS_D is a set of all tags in the XML document D.

Example 1.2.1 *(XML tree and DataGuide) Figure 1.1(c) shows the example of a DataGuide for the XML document in Figure 1.1(a), where the* $TGS_D = \{a, b, id, c, d\}$. *The XML tree contains eight labeled paths, where some of them are* $a/b/c$, $a/b/d$ *or* $a/b/d/b/d$. *The node* d_2 *corresponds to labeled path* $a/b/d/b/d$, *where the labeled path* $a/b/d/b/d$ *is of class* d.

1.3 Problem Statement

A number of special query languages like *XPointer* [63], *XQL* [51], *XPath* [62], and *XQuery* [60] have been developed to obtain specified data from an XML database. These languages use the concept of *regular path expression* (RPE) to specify a path in the semi-structured data. The more general form of RPE is called Twig Pattern Query (TPQ) and the problem of TPQ matching has become a challenging research topic for many works [1, 5, 27].

1.3.1 Twig Pattern Query Matching

In Figure 1.2 we introduce the grammar of TPQ used in this thesis. TPQ can be modeled as an unordered rooted *query tree*, where each node of the tree corresponds to one *query node* and edges represent a descendant or child relationship between the connected nodes. Each query node can be constrained by a value-based condition.

Since each query node is labeled by a tag, we use the term 'query node' in the meaning of 'tag of the query node'.

TPQ	$\rightarrow$	Axis Step Path
Axis	$\rightarrow$	`/` | `//`
Path	$\rightarrow$	Axis Step Path | ϵ
Step	$\rightarrow$	TAG Predicates
Predicates	$\rightarrow$	[Andexpr] Predicates | ϵ
Andexpr	$\rightarrow$	Orexpr AND Andexpr | Orexpr
Orexpr	$\rightarrow$	Notexpr OR Orexpr | Notexpr
Notexpr	$\rightarrow$	NOT Eqexpr | Eqexpr
Eqexpr	$\rightarrow$	TPQ Valpred VALUE
Valpred	$\rightarrow$	$= \mid < \mid > \mid \geq \mid \leq$

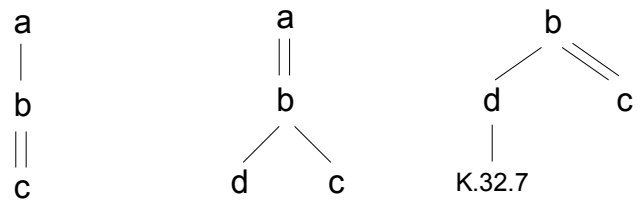
Figure 1.2: Gramar of TPQ

We introduce some terms related to TPQ as follows:

- **Path query pattern p_q in a TPQ** is a root to the q node sequence of query nodes in the TPQ.
- **Node n of class q** is a node in an XML tree with the same tag as the query node q.
- **Labeled path lp of class q** is a labeled path, where the last tag of the lp labeled path is the same as the tag of the q query node.
- **Query match of a TPQ Q in an XML tree D** is a mapping from query nodes in a Q to nodes in D such that:
  - matched nodes in an XML tree satisfy the query node predicates,
  - relationship between the matched nodes satisfies the relationship between the corresponding query nodes.

- **Solution** of a path query pattern p_q is a sequence of nodes from the XML tree matching the p_q.

Example 1.3.1 *(Examples of TPQ)*



(a) (b) (c)

Figure 1.3: (a) Simple path query (b) TPQ (c) content-based TPQ

Figure 1.3(a) shows a query tree for the TPQ Q_1 `/a/b//c`*, Figure 1.3(b) shows the query tree for the TPQ* Q_2 `/a//b[./d]/c`*, and the query tree for the TPQ* Q_3 `//b[./d = 'K.32']//c` *is in Figure 1.3(c). Figure 1.4 shows the query matches of the TPQ* Q_2 *in the XML tree in Figure 1.1(b).*

One possible solution for the path query pattern `/a//b/c` *is the sequence* (a_1, b_1, c_1)*. In other words, the solution represents one path in a query match.*

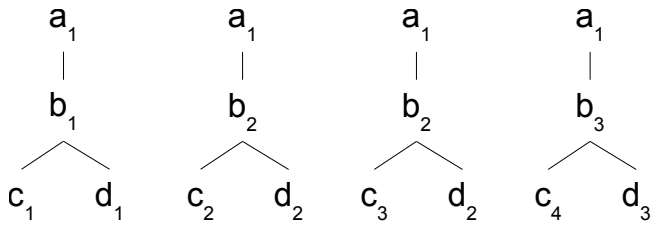


Figure 1.4: Query matches

Twig query matching is addressed by many works. However, query matches are usually not the expected output of an XPath query. We expect that only the nodes corresponding to the last location step of an XPath query are in the result set. This issue is more complicated in the case of XQuery, where the output can be defined more precisely. Consequently, we define the expected output of a TPQ slightly differently to other holistic approaches [1, 5, 27].

Definition 1.3.1 *(Result Set of a TPQ) The result set of a TPQ in an XML tree is an ordered set of nodes n_i of class q_{output}, where for every node n_i there exists at least one query match and the q_{output} is a query node corresponding to the last location step.*

Example 1.3.2 *(The Result of TPQ) The result set of the TPQ* `/a//b//c` *in the XML tree in Figure 1.1(b) is $\{c_1, c_2, c_3, c_4\}$. On the other hand, the query matches of the same query are $\{(a_1, b_1, c_1)$, (a_1, b_1, c_2), (a_1, b_2, c_3), (a_1, b_2, c_2), (a_1, b_3, c_3), $(a_1, b_3, c_4))\}$. We can observe that nodes of class c are duplicated in this query match set. Their occurrences are not even sorted although the solutions are sorted. However, according to an XPath we need nodes of class c without duplication in the result set, and therefore we must extract them from the query matches.*

Retrieval of the result set is more convenient compared to the query matches. We describe this issue in Section 2.2 more thoroughly. This difference is not significant for our results. We can always return results in a form of the query match, but the result set can simplify the TPQ processing in some cases and in the case of XPath language is a requested result.

Due to this difference we refer the TPQ Q result set retrieval as *TPQ searching*.

1.3.2 Labeling Schemes

The labeling scheme associates every node in the XML tree with a label. These labels allow to determine structural relationship (reachability) between nodes.

Structural relationship between nodes can be also resolved using the navigation [40, 39] in the XML tree. This method usually requires traversal of many nodes in a subtree which are irrelevant to the query. Due to this fact this method is not considered here and we will focus only on labeling schemes.

We can find many different labeling schemes with various properties. Every labeling scheme allows to resolve basic relationship between nodes (following, preceding, ancestor, descendant). We recognize two main types of labeling schemes:

- **Element labeling scheme** - Labels are represented by the fixed length codes allowing to resolve basic relationship between two nodes by comparing fixed number of label values. Labels are usually stored together with node level information in order to resolve the parent–child relationship. Every node is identified by its first label value (we call it *nodeid*). We can attach parent node *nodeid* to a node label which allows us to resolve the following–sibling and preceding–sibling relationship.

- **Path labeling scheme** - Labels are represented by the variable length codes, where we can extract labels of all ancestors of a node. We have to compare whole prefix of a nodes' labels in order to resolve relationship between two nodes. Labels under path labeling schemes do not need other informations to resolve the parent–child and sibling relationship.

Example 1.3.3 *(Element Labeling Scheme) Figures 1.5(a) and 1.5(b) show the XML tree labeled by* Containment labeling scheme [71] *and* Dietz labeling scheme [21]*, respectively.*

The Containment labeling scheme creates labels according to the document order. We can use simple counter, which is incremented every time we visit an start or end tag of an element. The first and the second number of a node label represent a value of the counter when the start tag and the end tag are visited, respectively.

The Dietz labeling scheme assigns the first number according to the document order of the element (preorder) and the second number according to the postorder.

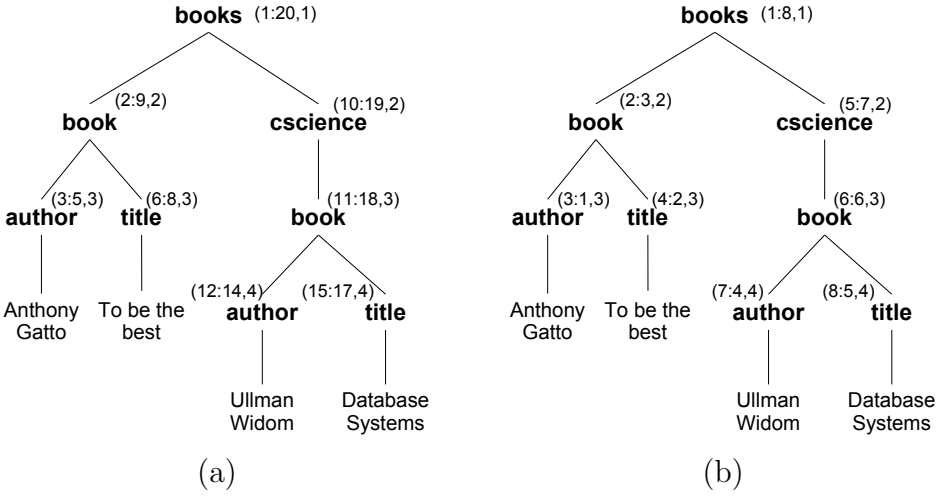


Figure 1.5: (a) Containment labeling scheme (b) PrePost labeling scheme

Example 1.3.4 *(Path labeling scheme) Figures 1.6(a) and 1.6(b) show the XML tree labeled by a* Dewey order [56] *and* OrdPath labeling scheme [44]*, respectively. In the case of Dewey order every number in the label corresponds to one ancestor node. In the case of OrdPath only odd numbers correspond to ancestor nodes and the even numbers serve as prefixes of the odd numbers. For example, let us have the label*

(1,3,2,1,1) of node **`author`** *in Figure 1.6(a). The first number corresponds to* **`books`** *node and the second number corresponds to* **`cscience`** *node, but the third number is even. Therefore, the fourth number corresponds to* **`book`** *node.*

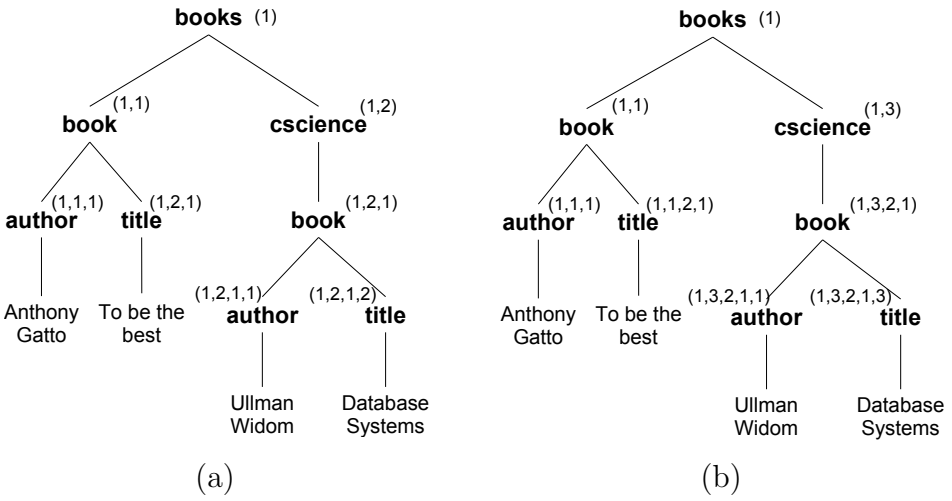


Figure 1.6: (a) Dewey Order labeling scheme (b) OrdPath labeling scheme

Major differences are in the field of updating of an indexed XML document. We can relabel the whole XML tree when we use element labeling scheme and when we want to insert into this tree. The Dewey order can accommodate newly inserted nodes well [44] and in the worst case we have to relabel the subtree, where we insert. The OrdPath labeling scheme does not need to update labels during the insertions at all. However, this unique feature is balanced by occurrence of long labels after many insertions. We sometimes have to relabel a tree to decrease the labels' length in the case of OrdPath.

1.4 Related Work

We can find many various approaches to a TPQ searching. First we can distinguish two sets of approaches, depending on whether they do not use some precomputed labeling scheme or they do use it [20].

The first set of approaches is mainly represented by the work of Gotlob et al. [17, 18, 19] or by Michael Kay's Saxon application [41]. In this case the query processing

is closer to streaming approaches. They almost sequentially read the input document during the query processing. These approaches store parts of an XML document in main memory, and therefore they are not suitable for querying large XML documents.

The approaches of the second set represent more database-like solutions. XML data are first preprocessed and then stored in an RDBMS or in special purpose data structures. The queries are then processed with a support of an RDBMS or data structures. There is a considerable amount of work done in this area during the last ten years. Approaches address various aspects of the XPath and XQuery languages. We classify approaches based on a labeling scheme to be better acquainted with their features as follows:

- **RDBMS approaches** - these approaches use the RDBMS as a data storage and a query processor. The XML document is decomposed into the relations and any TPQ query is translated into the SQL query. We further differentiate the RDBMS approaches depending on a way of an XML document storage as follows:
  - *Schema aware approaches* [54, 12] (inlining approaches) use kind of horizontal decomposition of an input XML document. The sibling elements are usually stored in the same record in the relation. Approaches utilize DTD or XML schema in order to create a set of relations for the XML data.
  - *Schema oblivious approaches* [16, 56, 10, 21, 38, 11] decompose an input XML document into a set of relations, where each XML tree node is stored in separate record. Every record stores mainly node label.
- **Native approaches** do not use the RDBMS query processor for a TPQ processing. However, these approaches can possibly use the RDBMS as a storage system for the labeled data. Similarly to schema oblivious methods, they store one node label per record. The inverted list can be also utilized by these approaches. We can divide native approaches into two main categories:
  - *Structural join approaches* [34, 1, 71] propose a new type of TPQ query operation utilizing some labeling scheme. This operation is performed between two sets of nodes corresponding to some query node. The TPQ processing then consists of a set of structural joins.
  - *Holistic algorithms* [5, 9, 36, 8] handle TPQ processing as one join operator. Multiple input streams are joined during their simultaneous scan.

In the following section we describe every above mentioned category in detail.

1.4.1 Schema Aware RDBMS Approaches

Inlining approaches [54, 12] use XML schema or DTD to create appropriate set of relations which are connected together using foreign keys. An example of relations created for XML document in Figure 1.9(a) can be seen in Figure 1.9(b).

```
<?xml version="1.0" ?>
<a>
  <f>text</f>
  <b>
    <c>text1</c>
    <d>text2</d>
  </b>
  <b>
    <d>text4</d>
    <c>text3</c>
    <e>text5</e>
  </b>
</a>
```

(a)

Figure 1.7: element a

nodeid	parent id	f
1	null	text

Figure 1.8: element b

nodeid	parent id	c	d	e
2	1	text1	text2	null
3	1	text3	text4	text5

(b)

Figure 1.9: Example of an XML document decomposed using inlining

The main advantage of these approaches is a possibility to fully utilize features of an existing RDBMS. Many RDBMS provides transactions, value indices, parallelization, distribution, and other techniques used in relational databases. However, utilizing RDBMS query planer can lead to a bad performance and query processing workload can be very high. The RDBMS query evaluation using nested loop joins can work very poorly as shown in [71].

Approaches based on this type of indexing are also constrained by the schema of the XML document. If the schema is not available, we have to analyze the input XML document and create an appropriate schema for this document. There is also another problem with sparse relations since the relations cover all possible siblings within the particular node. These siblings are not always presented in every node and the relation then contains an empty space as can be seen in the relation 1.8 for the element `b`.

1.4.2 Schema Oblivious RDBMS Approaches

There can be found number of proposals using XQuery-to-SQL mapping with XML data stored in a relational database [16, 56, 10, 21, 38]. XML tree nodes are labeled

by some labeling scheme and stored in the Edge table. Nodes in the Edge table are clustered according to tag name [16, 56, 21] or according to labeled path [38, 11].

The processing of XML queries is performed by a translation of an XML query into a SQL query. These methods utilize properties of a different labeling schemes in order to speed-up SQL queries. The attributes of the PrePost labeling scheme are used in [21, 22] and the Dewey labeling scheme is used in [56]. Methods utilize various techniques in order to prune the maximum of irrelevant nodes before the nested-loop join of the SQL query.

1.4.3 Structural Joins Approaches

These works can use inverted list as an underlying data structure for a node retrieval. Similarly to schema oblivious approaches XML tree nodes are labeled by some labeling scheme and stored in a inverted list.

Work [71] shows that the number of joins during a query processing can be significantly reduced if the RDBMS nested-loop join is not used. Proposed solution is based on a MPMGJN join algorithm. The binary structural join was soon identified [1] as an elementary operation for a TPQ processing.

The TPQ can be processed using structural joins by various query plans. Work [68] proposes a query plan selection for the structural join algorithms. These methods use some result estimation method, e.g., [67] in order to select the best query plan.

Structural join approach is the core method of the TIMBER project at University of Michigan [57].

1.4.4 Holistic Approaches

Structural joins can produce large intermediate results, even though the final result is small. To overcome this problem, holistic approach has been proposed [5] handling a TPQ as a single operator. The authors of [5] propose the TwigStack algorithm, which is shown to be optimal for TPQ having only the ancestor-descendant relationship (we refer this type of query as the AD query in the following text). In the case of the AD query the TwigStack finds only solutions contributing to some query match. If a TPQ has edges with a parent-child relationship, algorithm find solutions which do not contribute to any query match and we have to prune these in the second stage of the algorithm. To decrease the whole number of irrelevant solutions in the case of the PC edges work [35] proposes a look-ahead approach.

Work [9] proposes a more general version of the TwigStack algorithm allowing utilization of various keys (streaming schemes) in an inverted list. They test tag,

tag+level, and labeled path as a key of an inverted list. There are also works utilizing path labeling scheme for a TPQ processing [36]. This approach allows to handle only leaf query nodes during the query processing.

1.4.5 Indexing Methods

The nodes in inverted list can be further indexed. We distinguish two types of indices; the *tree structural index* and the *value index*.

Structural index can be involved in most approaches using some labeling scheme. Value index represented by a B^+-tree can be involved in any approach described above.

The tree structural index for element labeling scheme is discussed in [5]. This work proposes slight extension of a B^+-tree called XB-tree. There are also other tree structural indices allowing to skip some elements in the stream. The comparison of tree structural indices can be found in [33]. General conclusion of this work is that the XB-tree outperforms other works for highly recursive documents and the results are comparable for non-recursive documents.

1.4.6 Additional Comparison of Approaches

There are also other approaches to TPQ processing [65, 50]. In [42] we can find a comparison of many different approaches to TPQ processing. The holistic approaches are considered as a most robust solution which does not need sophisticated query optimizations.

This is an important conclusion because we are utilizing the holistic algorithms as well in this thesis.

1.5 Contribution

In this thesis we make a work toward an improvement of indexing methods and holistic algorithms to a twig pattern query searching. We analyze the state-of-the-art approaches and propose their additional improvement. We focus mainly on approaches using labeled paths and path labeling scheme.

- We propose novel strong independence condition and prove optimality for the holistic algorithms when our novel condition is satisfied.

- We propose two holistic algorithms using labeled paths being optimal when the strong independence condition is satisfied. In the comprehensive experiments we show the advantages of our new algorithms.
- Algorithms using labeled paths utilize the stream pruning technique as a first step of the query processing. We propose methods for the stream pruning algorithm utilizing existing holistic algorithms. We design fast enumeration algorithm for this purpose.
- We describe what has to be satisfied by an XML document in order to satisfy strong independence condition for any TPQ.
- We describe multi-dimensional index and relational index supporting two types of structural joins. We propose a cost-based join selection algorithm capable to select optimal query plan. The algorithm consider appropriate type of the join operation during the TPQ processing.

Chapter 2

Holistic algorithms

2.1 Holistic Algorithms Basis

In this chapter, we review holistic algorithms for TPQ searching outlined in Section 1.4.2. There are many works describing various holistic algorithms [5, 28, 26, 9, 36, 8, 70]. Some of them may be combined, e.g. XB-tree can be utilized in all holistic approaches applying an element labeling scheme.

Some holistic algorithms describe processing of specific operators like 'or' or 'not' [26, 70]. We do not include their description here, but any top-down holistic processing can be easily extended if we include description in [26, 70]. The optimality of a holistic algorithm (we will describe this concept later) does not change if those operators are included.

We first introduce the notation used in all holistic algorithms. Supported twig query operations are the following: *isLeaf*: Query node → Bool, *parent*: Query node → Query node, *childs*: Query node → {Query node}, *subtreenodes*: Query node → {Query node}.

Holistic approaches use input streams, where every stream T_q belongs to the one query node q. We store node labels within the stream and nodes are always sorted in the stream. More than one input stream may be related to one query node. The supported stream operations are these: *head*: Stream → Node, *eof*: Stream → Bool, *advance*: Stream → Bool. We expect that the streams are supported by an efficient retrieval mechanism such as inverted list.

We often have to determine the nodes' relationship during the query evaluation. For this purpose, let us define the following operations: *ancestor*: Node × Node → Bool, *precede*: Node × Node → Bool. Furthermore, we say that the n_1 node is

smaller than the n_2 node iff n_1 is an ancestor or precedes the n_2 node.

In Section 2.1.1, we introduce a basic notation of different streaming schemes used by holistic algorithms. In the works TwigStack [5], iTwigJoin [9], and TJFast [36], authors provide some query classes for which the algorithms are optimal. Both space and I/O complexities of the optimal algorithm can be determined and the space complexity is low. Algorithms such as bottom-up Twig^{2}Stack [8] have the space complexity rather high and due to this fact we do not consider these approaches. The Twig^{2}Stack algorithm can even load the whole document into the main memory. Therefore, only top-down algorithms are briefly described in Section 2.2. In Sections 2.1.2 and 2.2.3, we summarize the optimality of holistic algorithms.

2.1.1 Stream Pruning

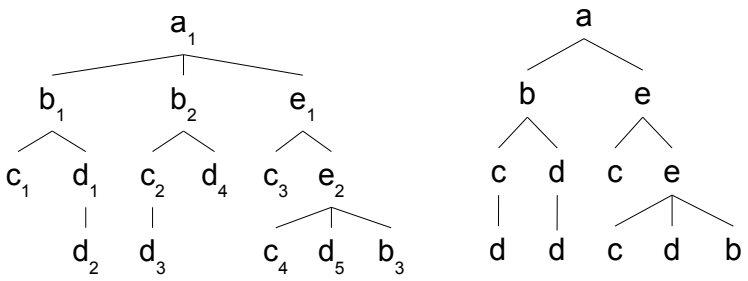
The TwigStack algorithm uses the *Tag* streaming scheme. In [9], Chen et al. propose an extension of this algorithm by using a different streaming schemes and they introduce *Tag+Level* and *Prefix path* streaming schemes. The Prefix path streaming scheme is basically a utilization of labeled paths. The idea of having a labeled path as a key for node retrieval has already been introduced in previous works [38, 55, 30]. However, Chen et al. were the first who described a holistic algorithm using the labeled paths (Prefix path streaming in their language). Depending on a key for a node retrieval we classify the approaches on element-based approaches and path-based approaches [7].

In the following text, we mainly focus on labeled paths. We do not consider the Tag+Level streaming scheme, mainly due to the fact that its filtration mechanism is not sufficient [9]. We use LPS instead of 'Labeled path streaming scheme' or 'Prefix path streaming scheme' in the following text.

It is important to specify the relation among streams in the context of a TPQ. In [9], authors introduce the solution streams and stream pruning algorithm, which result in labeled paths matched by a TPQ. Consequently, this algorithm reduces the searched space.

Definition 2.1.1 *(Solution Streams) Solution streams $soln(T, q_i)$ for a stream T of class q and a q's child query node q_i in a TPQ are streams of class q_i which satisfy the structural relationship of an edge $\langle q, q_i \rangle$ with a T stream.*

In the following text we sometimes use labeled path lp instead of stream T_{lp}. Under LPS is the stream represented by the corresponding labeled path, and therefore these terms are equivalent.



(a) (b)

Figure 2.1: (a) An XML tree (b) A DataGuide of the XML tree

Example 2.1.1 *(Solution Streams) Let us consider the XML document in Figure 2.1(a) and the TPQ* `a//e[./d]//c`*. The labeled paths of class e are labeled paths* $\{a/e, a/e/e\}$*. Solution labeled paths for the labeled path* a/e *are defined as* $soln(a/e, d) = \emptyset$*,* $soln(a/e, c) = \{a/e/c,\ a/e/e/c\}$*. For the labeled path* $a/e/e$*, solution labeled paths are defined as* $soln(a/e/e, d) = \{a/e/e/d\}$*,* $soln(a/e/e, c) = \{a/e/e/c\}$*.*

An lp labeled path of class q may have a child query node q_{child} so that $soln(lp, q_{child}) = \emptyset$. In this case the lp is redundant and it can be removed from the labeled path set of the query node q. This labeled path is also removed from the $soln()$ multiset of its parent node. This is the main idea behind the stream pruning. For example, the a/e labeled path is pruned in Example 2.1.1. Consequently, we do not consider the $T_{a/e}$ stream during the twig pattern processing.

To specify the stream pruning in more detail, we use the definition of a U_T set used in [9]. The U_T set is defined as follows:

$$U_T = \begin{cases} \{T\} & \text{if } q \text{ is a leaf node,} \\ \{T\} \cup \{\cup_{q_i \in childs(q)} C_i\} & \text{if none of } C_i \text{ is } \{\}, \\ \{\} & \text{if one of } C_i \text{ is } \{\}, \end{cases}$$

where $C_i = \cup_{T_c \in soln(T, q_i)} U_{T_c}$

We obtain the $U_{T_{root}}$ set for every stream T of class q_{root}. The stream T' of class q is removed from the query node's stream set if there is not any $U_{T_{root}}$ such that $T' \in U_{T_{root}}$. A set of streams of class q which remains after the stream pruning is called PRU_q.

Lemma 2.1.1 *(Streams after Stream Pruning) For every stream $T \in PRU_q$, where $q \neq root$, there exists at least one stream T' of class q_{parent}, where $T \in soln(T', q)$.*

Lemma 2.1.1 follows the stream pruning definition. For any stream $T \in PRU_q$ there exists a $U_{T_{root}}$ such that $T \in U_{T_{root}}$. Hence we see that there exists a stream $T' \in U_{lp_{root}}$ of class q_{parent} such that $T \in soln(T', q)$.

2.1.2 Optimality Basis

In this section, we introduce a basic notion of the optimality of holistic algorithms.

The important feature of a holistic algorithm is that it always works only with the head node of the streams. We can store some nodes in the stacks, however, the head nodes are the only nodes which are accessible from the stream during the each stage of the query processing.

In the following definition, we use a query match of a Q_q (a subtree of Q rooted by q), which has the similar meaning as a query match of a Q. The only difference is that for a query match of a Q_q we consider query nodes only from the $subtreenodes(q)$.

Definition 2.1.2 *(Minimal Extension) Let us have a TPQ Q. The node $n = head(T_q)$, $q \in Q$, has a minimal extension if there is a query match of a subtree Q_q, where every node of the query match is the head node of its stream.*

Example 2.1.2 *(Minimal Extension) Let us consider the XML document in Figure 2.1(a) and the TPQ Q `/a/b[./c]//d`. If we use the Tag streaming scheme, we have four streams corresponding to the query nodes in the TPQ Q. At the beginning of the algorithm the streams' heads are as follows: $head(T_a) = a_1$, $head(T_b) = b_1$, $head(T_c) = c_1$, and $head(T_d) = d_1$. The node b_1 has the minimal extension (b_1, c_1, d_1) which is the query match of the subtree Q_b. Also the node a_1 has the minimal extension (a_1, b_1, c_1, d_1), which is the query match for the whole query.*

We also define a *possible match* of a subtree Q_q as follows: it is a query match of a subtree Q_q that is composed of the $head(T_q)$ nodes or nodes belonging to the $tail(T_q)$. At least one $head(T_q)$ has to be a part of the possible match of a subtree.

Example 2.1.3 *(Possible Match) Let us have the XML document in Figure 2.1(a), the TPQ Q `/a/b[./c]/d` and we consider the Tag streaming scheme again. After few iterations of a holistic algorithm we get into the configuration, where the $head(T_a) = a_1$, $head(T_b) = b_2$, $head(T_c) = c_2$, and $head(T_d) = d_3$. There are two possible matches for the Q_b subtree: $(b_2, c_2, tail(T_d))$ or $(tail(T_b), tail(T_c), d_3)$, and there is no minimal extension for the node b_2.*

Optimality of a holistic algorithm depends on a streaming scheme utilized as shown in [9]. We use their definitions of blocked, useless, and matching node. We distinguish among three types of head nodes of a stream T_q with respect to a Q:

- *matching node* – a node n of class q is called a matching node if n has a minimal extension and it is not in a possible match of a subtree $Q_{q_{parent}}$,
- *useless node* – a node n of class q is a useless node if there is no possible match of a subtree Q_q with the n node,
- *blocked node* – otherwise n is a blocked node.

In other words, useless node n of class q is a node which does not participate in any query match. Therefore, T_q can be advanced and the node n can be skipped. The matching node has a query match in its subtree and we can decide if the node is in a query match of a Q by using a simple mechanism. In holistic algorithms, such a mechanism is represented by stacks.

We say that a holistic algorithm is optimal when we can always find a stream with a head node that is either matching or useless.

2.2 Top-down Holistic Approaches

In this section, we focus on top-down algorithms for the TPQ searching without an additional index support. These algorithms apply various streaming schemes (e.g., Tag streaming schemes or LPS) and various labeling schemes (e.g., element or path labeling schemes). Streaming and labeling schemes corresponding to considered holistic algorithms are as follows:

- TwigStack [5]: tag streaming scheme and element labeling scheme,
- iTwigJoin [9]: any streaming scheme and element labeling scheme,
- TJFast [36]: tag streaming scheme and path labeling scheme.

The iTwigJoin is a more general version of the TwigStack, and therefore we only consider the iTwigJoin algorithm. In the following text, we refer to the TwigStack algorithm as the iTwigJoin algorithm with the Tag streaming scheme.

All top-down holistic algorithms use one stack S_q per query node q. Every item on the stack has only ancestors below itself during the query evaluation. Each stack supports common stack operations: *pop*, *push*, *top*, and *empty*.

Every top-down holistic approach operates in two phases. In the first phase, solutions of a query path from a TPQ are found and stored in output arrays. In the second phase, solutions in output arrays are merged and query matches are found. The number of output arrays is equal to the number of leaf nodes in a query.

2.2.1 iTwigJoin

The iTwigJoin algorithm uses an element labeling scheme no matter which streaming scheme is used. The algorithm repeatedly calls the *getNext* function during the first phase. The *getNext* function returns a query node $q \in Q$. The $head(T_q^{min})$ is consecutively pushed to the stack S_q if it has a matching node in the parent stack. Hence we know, whether the $head(T_q^{min})$ is in the query match of Q or not. Other features of this algorithm are described in Section 2.2.3.

Algorithm 1: iTwigJoin

```
1  pruning(TPQ);
2  while ¬end do
3      q = getNext (root);
4      T_min = min(q);
5      pop(S_parent(q), head(T_min));
6      if isRoot(q) ∨ ¬S_parent(q).empty() then
7          pop(S_q,head(T_min));
8          S_q.push(head(T_min));
9          if isLeaf(q) then solutionWithBlocking(S_q);
10     end
11     advance(T_min);
12 end
```

function: pop(Stack S, Node e)

```
1  while ¬S.empty() and ¬ancestor(S.top(),e) do
2      S.pop();
3  end
4  mergePhase();
```

The correctness of the iTwigJoin algorithm has been shown by Chen et al. in [36].

Example 2.2.1 *(iTwigStack+Tag)*

Let us consider the XML tree in Figure 2.2(a) and the TPQ Q_1 `//a//b//c`*. Table 2.2(b) shows the situation during each cycle of the TwigStack algorithm. The second column shows the node returned by the* $getNext()$ *method and the third column*

Algorithm 2: getNext(q)

```
1  if isLeaf(q) then return q;
2  foreach q_i ∈ childs(q) do
3      q_x = getNext(q_i);
4      if not q_x = q_i then return q_x;
5  end
6  foreach Stream T_q^j of class q do
7      foreach q_i ∈ childs(q) do
8          T_{q_i}^{min} = min(soln(T_q^j,q_i));
9      end
10     T_max = max({T_{q_1}^{min}, ..., T_{q_n}^{min}});
11     while precede(head(T_q^j),head(T_max)) do
12         advance(T_q^j);
13     end
14 end
15 T_q^{min} = min(streams(q));
16 T_{child}^{min} = stream with smallest node among all streams in child query nodes;
17 if head(T_q^{min}) < head(T_{child}^{min}) then return q;
18 return q_c where the T_{child}^{min} is of class q_c;
```

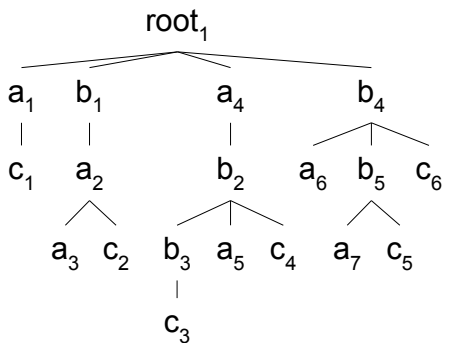
depicts the streams' head nodes at the beginning of the cycle. The operations on the stacks are shown in the third column.

In the second cycle the getNext() function return the node b_1 with the minimal extension (b_1, c_2), however, the node is useless because the parent stack is empty. The first node matching the query is the node a_4 with minimal extension (a_4, b_2, c_3). Due to the fact that the algorithm is optimal for this query (we describe the algorithm optimality closer in Section 2.2.3) every node of class c pushed into the stack S_c is also stored in the result set. Therefore, we get the TPQ Q_1 result set $\{c_3, c_4\}$.

In the case of a non-optimal algorithm (optimality of algorithm usually depend on a query), we have to store the solutions in the output arrays and prune irrelevant solution paths during the second phase (merge phase). As shown in [35] the number of irrelevant stored solution paths may be significant.

2.2.2 TJFast

The TJFast [36] algorithm differs from the previous algorithm mainly in the labeling scheme used. In [36] authors introduce a type of path labeling scheme called *Extended Dewey* labeling scheme.



(a)

Cycle	Selected Node	Head Nodes	Operations
1	c_1	a_1, b_1, c_1	useless node
2	b_1	a_1, b_1, c_2	useless node
3	c_2	a_2, b_2, c_2	useless node
4	a_4	a_2, b_2, c_3	push
5	b_2	a_5, b_2, c_3	push
6	b_3	a_5, b_3, c_3	push
7	c_3	a_5, b_4, c_3	push
8	c_4	a_5, b_4, c_4	pop b_3, pop c_3, push
9	b_4	a_5, b_4, c_5	pop a_4, pop b_2, pop c_4
10	b_5	a_6, b_5, c_5	useless node
11	c_5	$a_7, -, c_5$	useless node
12	c_6	$a_7, -, c_6$	useless node

(b)

Figure 2.2: (a) Example of an XML tree (b) TwigStack processing of the TPQ Q_1

Finite state transducer

FST is an oriented graph $G(V, E)$, where every *FST node* $n \in V$ corresponds to one node tag and every *FST edge* $e \in E$ is labeled by a number e_{num}. The number e_{num} is unique among all FST edges coming from the same FST node. We assign n_{max} value to every FST node n, where $n_{max} > e_{num}$ for all FST edges e coming from n.

There is a method of an Extended Dewey encoding which allows the labeled path extraction. This is possible using the *Finite State Transducer* (FST).

The Extended Dewey label for a node n is a tuple $(l_1, \ldots, l_m)$, where the m is the level of a node n in an XML tree. Using an FST and the label of a node we can obtain the labeled path:

1. We start from the FST node which corresponds to the document root node tag (the node's tag becomes the first item in the labeled path),

2. k = 1,

3. We search for an edge e coming from an actual FST node n, where $l_k \ mod \ n_{max} = e_{num}$,

4. Finding this edge, we move to the next FST node (we add its tag to the labeled path),

5. k = k + 1. If k > m, then the labeled path extraction ends, otherwise we continue with the step 3.

Example 1 (Extended Dewey).

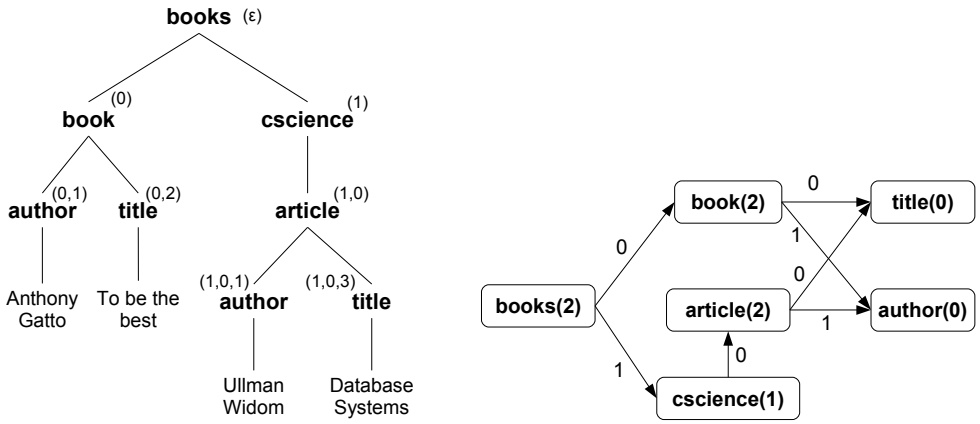


Figure 2.3: (a) XML tree labeled by an Extended Dewey labeling scheme (b) corresponding FST

In Figure 2.3(a) we can observe an example of an XML tree labeled by an Extended Dewey labeling scheme. The FST corresponding to the XML tree is in Figure 2.3(b). Each FST node n is labeled by a tag and its n_{max} is in parentheses.

If we want to know the labeled path of a node with the $(1, 0, 3)$ label, we start from the `books` FST node (step 1). This is also the first tag of the labeled path. We search for a edge with number 1 ($l_1 = 1$, $n_{max} = 2$). Clearly, edge with the number 1 points to the `cscience` FST node. We append this value to the labeled path and continue. The next search goes directly to the `article` FST node and during the last search we find an edge with the number 0 ($l_3 = 2$, $n_{max} = 2$). This edge point to the `title` FST node and this is the last labeled path tag.

This extraction is time consuming due to the fact that it is often called during the query processing. In Section 2.4.2 we introduce the TJDewey algorithm that utilizes LPS and which allows us to omit this time consuming labeled path extraction.

TJFast Algorithm

If we compare the TJFast to other holistic algorithms, we distinguish one important improvement: a query is evaluated with a retrieval of nodes from the leaf query node streams. For example, having the query `a//e/c`, we find solutions with the retrieval of Extended Dewey labels from the `c` stream.

In Algorithm 3 we can observe the main cycle of the TJFast algorithm. This algorithm shares the $getNext()$ (Algorithm 4) function with the TJDewey algorithm described in Section 2.4.2. The result nodes are influenced by the nodes on the stacks (the *outputSolutions()* method). We use the $dbl(q)$ function in these algorithms representing the set of direct descendant branching or labeled query nodes.

The TJFast use MB set, which is created for every node label. It store ancestors of the label according to the query path. We explain the MB set more thoroughly in Section 2.4.2.

Algorithm 3: TJFast algorithm

```
1 while ¬end do
2     q = getNext (root);
3     outputSolutions(q);
4     advance(T_q);
5     locateMatchedLabel(q)
6 end
7 mergePhase();
```

procedure: locateMatchedLabel(Query node q)

```
1 while head(T_q) do not match the pattern p_q do
2     advance(T_q);
3 end
```

function : outputSolutions(Query node q)

1 Return solutions of `head(`T_q`)` related to p_q such that the nodes matching the p_q exist in the ancestors' stacks.

When we have a labeled path corresponding to node n of class q_{leaf}, we can decide if the labeled path matches the path query pattern p_q in procedure *locateMatchedLabel()* using some string matching algorithm. Therefore the TJFast work only with labels matching its query path and the TJFast only resolve whether the label match the whole TPQ or not.

Algorithm 4: *getNext(q)* function

```
1  if isLeaf(q) then return q;
2  foreach q_i ∈ dbl(q) do
3      f_i = getNext(q_i);
4      if not isBranching(f_i) ∧ empty(S_{q_i}) then
5          clearSet(q, T_{q_i}^{min});
6          return f_i;
7      end
8      e_i = max{p | p ∈ MB(f_i,q)};
9  end
10 max = maxarg_i{e_i};
11 min = minarg_i{e_i};
12 foreach q_i ∈ dbl(q) do
13     if ∀p ∈ MB(f_i,q): ¬ancestor(p, e_max) then
14         clearSet(q, T_{q_i}^{min});
15         return q_i;
16     end
17 end
18 foreach p ∈ MB(q_min,q) do
19     if ancestor(p, e_max) then
20         updateSet(S_q,p);
21     end
22 end
23 return q_min;

   function: clearSet(Stack S_q, Node p)
1  if ¬ancestor(S_q.top(),p) then
2      S_q.pop();
3  end

   function: updateSet(Stack S_q, Node p)
1  clearSet(S_q, p);
2  S_q.push(p)
```

2.2.3 Optimality of Holistic Algorithms

Following Section 2.1.2, we say that a holistic algorithm is optimal when we can always find a stream with matching or useless head.

Every holistic algorithm has its own specific classes of queries, where the optimality is proved.

We precisely specify the query classes being optimal for at least one holistic algorithm. The following definitions express what have to be satisfied by the queries belonging to the query class.

- *AD query* - query containing only AD edges, where query may begin with the PC edge.
- *PC query* - query containing only PC edges, where the query may begin with the AD edge.
- *One branching node query* - query containing maximum one branching node.
- *AD in branches query* - query containing only AD edges after first branching node.
- *PC before AD query* - query where the AD edge is never before the PC edge in the query.

The examples of the query classes can be seen in Figure 2.4.

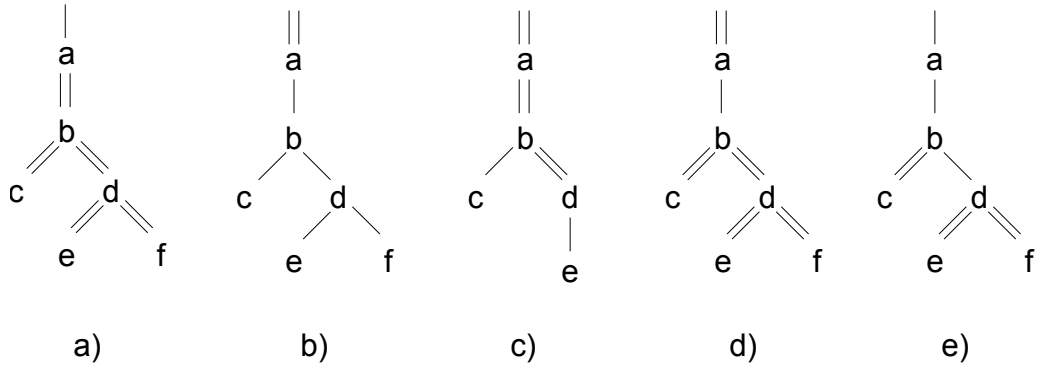


Figure 2.4: (a) AD query (b) PC query (c) One branching node query (d) AD in branches query (e) PC before AD query

An algorithm optimality depends on the streaming scheme and the query class. The query classes proved in previous works [5, 9, 36] are as follows:

- TwigStack [5] – an AD query,
- iTwigJoin+TL [9] – an AD query or PC query,
- iTwigJoin+LP [9] – an AD query, PC query, or one branching node query,
- TJFast [36] -an AD in branches query.

The attentive reader may notice that the PC before AD query class is not presented in this list. We discuss this query class in Section 2.3 and show that the iTwigJoin+LP is in fact optimal for this query class. This is one of the contributions of this thesis.

If an algorithm is optimal, the merge phase is skipped. This is slightly different from the description in previous works. Due to the fact that they expect the set of query matches as an output of an algorithm, the merge phase is processed even in the case of the optimal algorithm. We expect the output specified in Definition 1.3.1 to meet requirements of XPath. We store a node of the class q_{output} into the result set when this node is pushed into the stack. In the case of other holistic algorithms, the result set building is completely processed during the merge phase. Obviously, our result set building is applicable for *all* holistic approaches.

In the case of a non-optimal algorithm, we have to prune irrelevant solution paths during the merge-phase. We store solutions found during the first phase in a blocking structure. When we pop out the last item from the root stack, we enumerate blocked solutions. Blocked solutions are then stored in output arrays in a sorted order allowing for the fast merge phase. As shown in [35], the number of irrelevant stored solution paths may be significant. During the merge phase, we also have to avoid the duplication in query matches to retrieve the result set.

The worst-case space complexity of the optimal iTwigJoin and TwigStack is $|Q| \cdot L$, where $|Q|$ is the number of query nodes in the TPQ Q and L is the maximum length of the root-to-leaf path in the XML document.

The worst-case space complexity of the optimal TJFast is $|Q_L| \cdot L$, where $|Q_L|$ is the number of leaf query nodes in the TPQ Q.

The worst-case I/O and time complexity is equal to the sum of input stream sizes and size of the output list in the case of any optimal holistic algorithm.

2.3 Condition for a Holistic Algorithm Optimality

In the following text, we prove that if specific assumptions are met, then the holistic approach is optimal even for a query containing any combination of PC and AD edges with arbitrary number of branching nodes [B1].

In Section 2.3.1 we introduce optimality condition for a holistic approach using LPS. In Section 2.3.2 we introduce optimality condition for a Tag+Level streaming holistic algorithm and in Section 2.3.3 we define what has to be satisfied for an algorithm using the Tag streaming scheme.

2.3.1 Labeled Path Streaming Algorithm Optimality

Definition 2.3.1 *(Strongly Independent Labeled Path) Let lp be a labeled path of class q, where* $lp \in PRU_q$*. We say that the lp is strongly independent if there is no labeled path* $lp' \in PRU_q$*, where the* lp' *is an ancestor (or prefix) of the labeled path lp.*

Let us define new type of query node within the TPQ called *query node with a PC in a subtree*, which has a PC edge between itself and query nodes in the $subtreenodes(q)$. Clearly, the query node satisfying this condition can be only an inner query node. For example, the TPQ `/a//b[.//c/d]/f[.//h]//g//k` contains three query nodes with a PC in a subtree: a, b, c.

Lemma 2.3.1 *(A Child Relationship for Strongly Independent Labeled Paths) Let us have a TPQ Q where S is a set of query nodes with a PC in a subtree and labeled paths of class* $q \in S$ *are strongly independent. Let us have a query nodes q and* q_{parent} *in Q with the PC relationship. Then any node n corresponding to a labeled path* $lp \in PRU_q$ *having an AD relationship with a node* n' *corresponding to a labeled path* $lp' \in PRU_{q_{parent}}$ *has to have also the PC relationship with* n'*.*

Proof. We prove this lemma by a contradiction. Let us assume that there is a node n corresponding to an $lp \in PRU_q$ having an AD relationship with a node n' corresponding to a $lp' \in PRU_{q_{parent}}$, where query nodes q and q_{parent} have the PC relationship and the node n do not have the PC relationship with this node n'. Then, by Lemma 2.1.1, there exists a labeled path $lp'' \neq lp'$ belonging to $PRU_{q_{parent}}$. Obviously, the labeled path lp' has to be an ancestor of lp''. Since lp' and lp'' are strongly independent labeled paths (note that they correspond to a inner query node having a PC in a subtree), the lp' is not an ancestor of lp'' (see Definition 2.3.1),

which is impossible. □

It can be observed that if two query nodes q and q_{parent} are connected with a PC edge, then both have the same number of labeled paths after the stream pruning. It is caused by the fact that those labeled paths in query nodes have one-to-one relationship. Due to this simple observation we can say that the labeled paths of class q are recursive iff labeled paths of class q_{parent} are recursive. Obviously, the query nodes have to have different tag names.

Therefore, we do not have to check labeled paths of the query node q with a PC in a subtree if the parent edge is the PC edge. We call the remaining query nodes as a *checking nodes* of a query and we define the following TPQ operation, *checkingnodes*: Query → {Query node}. Only query nodes having the AD edge as a parent edge and having an PC edge in a subtree belong into the *checkingnodes*() set.

Example 2.3.1 *(Checking Nodes)*

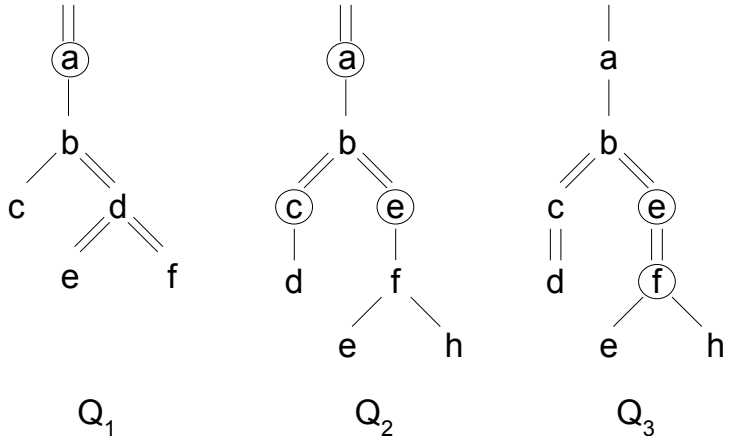


Figure 2.5: Example of three TPQs and their corresponding *checkingnodes*()

Figure 2.5 shows the example of three TPQs, where the checking nodes are in the circle. Considering the TPQ Q_1 `//a/b[./c]//d[.//e]//f` *we can see that the only checking node is the query node a since the second query node b with a PC in a subtree has a PC edge as a parent edge. Another example is the TPQ* Q_2 `//a/b[.//c/d and .//e/f[./g and ./h]]`*, where the checkingnodes*(Q_2) = {a,c,e}*. The last example shows the TPQ* Q_3 `/a/b[.//c//d]//e//f[./e]/h`*, where only the query nodes e and f are the checking nodes.*

We say that the *strong independence condition is satisfied for the TPQ Q and the XML document D* iff all the labeled paths of class $q \in checkingnodes(Q)$ are strongly independent.

We shall now prove the algorithm optimality when the strong independence condition is satisfied. The proof follows that presented in [9], which uses the mathematical induction. The major difference is in the base case, the induction step remains almost the same. Let us state the following definition which will be useful in our proof. By T_q^{min}, we denote a stream belonging to a query node q with the minimal node $head(T_q^{min})$ among all streams belonging to q. We call T_q^{min} the lowest stream in q.

Theorem 2.3.1 *(An Optimality for Strongly Independent Labeled Paths) Let us have an XML document D and a TPQ Q, where all labeled paths of class $q \in checkingnodes(Q)$ are strongly independent (i.e., the strong independence condition is satisfied). Then, under LPS, there always exists a head element which is useless or matching for Q.*

Proof. We use an induction on a subtree of Q:
(base case) Let us have a query node q which is a parent of leaf query nodes $q_1, \ldots, q_n$ and let T_q be an unfinished stream for the q. If there is a node $q_i \in childs(q)$ which does not have an unfinished stream, then all remaining nodes in T_q are useless and we are done. Obviously, every node $head(T_{q_i}^{min})$ has a minimal extension in Q_{q_i}.

The following cases can occur:

- if $head(T_q^{min})$ precedes $head(T_{q_i}^{min})$ for any $q_i \in childs(q)$, then the $head(T_q^{min})$ is the useless node for Q_q,
- else if $head(T_{q_i}^{min})$ is an ancestor or precedes $head(T_q^{min})$ for any $q_i \in childs(q)$, then the $head(T_{q_i}^{min})$ is the matching node for Q_q,
- otherwise $head(T_q^{min})$ is an ancestor of $head(T_{q_i}^{min})$ for every $q_i \in childs(q)$. By Lemma 2.3.1, every q_i satisfies the $\langle q, q_i \rangle$ relationship even in the case of the PC axis. Due to this, head nodes in $subtreenodes(q)$ create a minimal extension of q and $head(T_q^{min})$ is the matching node for Q_q.

(induction) The induction proceeds over any subtree Q_q. Let us have a query node $q \in Q$ with child query nodes $q_1, \ldots, q_n$, by the induction hypothesis, if there is a node $head(T_{q_i}^{min})$ which is not matching node for Q_{q_i}, we must find either useless node for Q_{q_i} (which is useless also for Q_q) or a matching node for Q_{q_i} (which is matching also for Q_q).

Otherwise, every node $head(T_{q_i}^{min})$ is a matching node for Q_{q_i} and we proceed with the same arguments as in the base case. The induction step ends when q is a root of Q and we can find matching or useless node for Q_q, which is therefore matching or useless node for Q and we are done. □

Theorem 2.3.1 proves the optimality of an algorithm using LPS if the strong independence condition is satisfied. Let us note that we work with the lowest stream of a query node in this proof. This means that we are able to find the matching or useless node only by checking the lowest stream in the query node if the strong independence condition is satisfied. This is important in the case of LPS algorithms introduced in Section 2.4.

Theorem 2.3.1 can be applied also to the case of the iTwigJoin+LP algorithm. In this way, we extend the set of queries for which the iTwigJoin+LP algorithm is optimal.

Example 2.3.2 *(Strong Independence Condition)*

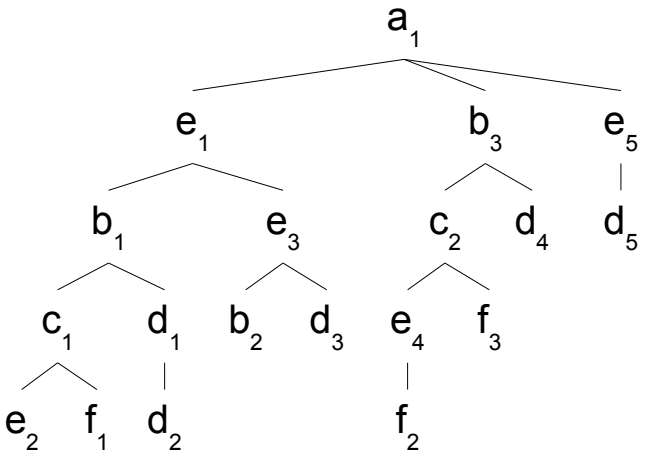


Figure 2.6: Example of the XML document

Let us consider the TPQ Q_4 `/a//b[.//c[./e and ./f]]//d` *and the XML tree in Figure 2.6, where* $checkingnodes(Q_4) = \{b, c\}$*. After stream pruning we get the following sets for each query node from* $checkingnodes(Q_4)$*:* $PRU_b = \{a/e/b, a/b\}$ *and* $PRU_c = \{a/e/b/c, a/b/c\}$*.*

In this case, all checking nodes contain only strongly independent labeled paths, and therefore a holistic algorithm using LPS is optimal for this query even though

the query does not belong to any query class depicted in Section 2.2.3.

The PC axes in the subtree Q_c `//c[./e and ./f]` *of the query Q_4 would cause the main problem when using the TwigStack algorithm. Let us have a configuration, where c_2, e_4, and f_2 are the head nodes of their streams T_c, T_e, and T_f. The f_2 node is not in minimal extension to c_2 node, but there can be such node in $tail(T_c)$. The c_2 node does not have any minimal extension, and thus all nodes are blocked in this subtree.*

If the strong independence condition is satisfied, then the above depicted situation does not occur during the query processing with an LPS approach. The labeled path $a/b/c/e/f$ corresponding to the node f_2 is pruned and therefore the algorithm using LPS is optimal.

Similar situation occurs in an LPS algorithm only if there are two labeled paths in inner query node, where one is a prefix of the other. For example, the TPQ Q `/a//e[./b and ./d]`*, where only the query node e is a checking node. After stream pruning we get the following set: $PRU_e = \{a/e, a/e/e\}$. The labeled paths in the PRU_e are not strongly independent, and therefore the algorithm is not optimal. Let us have a configuration, where the query nodes e_1, b_1, and d_3 are the head nodes of their streams T_e, T_b, and T_d. Obviously, all nodes are blocked, because the b_3 is a possible match and e_1 does not have a minimal extension. However, this query is optimal for iTwigJoin+LP algorithm. For details, see [9].*

A New Class of Optimal Queries

Recently proposed holistic algorithms always prove their optimality for a query class. We prove the holistic algorithm optimality based on characteristics of an XML document and not only for a query class. We further analyze this topic in Chapter 4.

Moreover, as a corollary of Theorem 2.3.1, we find a new class of queries where the algorithms using the LPS are optimal. We must keep in mind that the proposed strong independence condition is more general than this new query class.

Corollary 2.3.1 *(PC before AD query class) Using LPS, if the query Q has only PC edges before AD edges, then there always exists a head element which is useless or matching for a Q.*

If we have the PC edges before AD edges query Q, the $checkingnode(Q)$ have to be an empty set. Therefore, the strong independence condition is satisfied. Examples of such queries follow: `/a/b[.//c//d]//e`, `/a[.//b]/c` or `/a/b/c//d//e`. Queries which are not in this class are the following: `//a/b/c` or `/a[.//b/c]//d`. Other queries optimal for various holistic algorithms are depicted in Section 2.5.

2.3.2 Tag+Level Holistic Algorithm Optimality

Similarly to LPS holistic approaches we can find an optimality condition for a holistic algorithm using the Tag+Level streaming scheme.

We use the term *tag+level* representing one stream in Tag+Level streaming scheme similarly to the term labeled path in the LPS. We say that the tag+level tl is of class q, when the tag of tl has the same tag as the query node q.

Definition 2.3.2 *(Strongly Independent Tag+Level) Let the tag+level tl be of class q, where $tl \in PRU_q$. We say that the tl is strongly independent if the tl is the only one tag+level stream in the PRU_q.*

Let us define another type of a query node called thr *query node with a related PC edge*, which has a PC edge between itself and the query nodes in the $subtree(q)$ or have a PC edge between itself and its parent query node. For example, query `/a//b[.//c/d]/f[.//h]//g//k` contains five query nodes with a related PC edge: a, b, c, d, f.

Lemma 2.3.2 *(A Child Relationship for Strongly Independent Tag+Level) Let us have a TPQ Q, where S is a set of query nodes with a related PC edge and every tag+level tl of class $q \in S$ is strongly independent. Let us have a query nodes q and q_{parent} in Q with the PC relationship. Then any node n corresponding to a tag+tevel $tl \in PRU_q$ having an AD relationship with a node n' corresponding to a tag+level $tl' \in PRU_{q_{parent}}$ has to have also the PC relationship with n'.*

Proof. Both query nodes q and q_{parent} are query nodes with a related PC edge, and therefore PRU_q and $PRU_{q_{parent}}$ contain only one tag+level stream (see Definition 2.3.2). Due to this, if q and q_{parent} have the PC relationship, then the tag+level tl and tl' have to have the PC relationship, therefore any nodes $n \in tl$ and $n' \in tl'$ have the PC relationship. □

Theorem 2.3.2 *(An Optimality for Strongly Independent Tag+Level) Let us have a TPQ Q, where all tag+level streams corresponding to the query nodes with a PC in a subtree are strongly independent. Then, using the Tag+Level streaming scheme, there always exists a head element which is useless or matching for Q.*

The proof of Theorem 2.3.2 is analogous to that of Theorem 2.3.1. Note that the optimality of an algorithm is driven mainly by the strong independence definition

for the streaming scheme. The definition of the strong independence in the case of Tag+Level streaming is more restricted then in the case of LPS. This means that there can exist a TPQ Q which satisfies the strong independence condition under LPS, but not under Tag+Level streaming.

Example 2.3.3 *(Comparison of the LPS and Tag+Level Streaming) The TPQ* Q_5 `/a//b[.//c[./e and ./f]]//d` *in Example 2.3.2 satisfies the strong independent condition for the XML tree in Figure 2.6 under LPS. But this is not true for Tag+Level streaming, because query nodes with a related PC edge* (a, b, c, d, e, f) *contain more than one stream per query node after stream pruning. For example, the query node* b *contains streams* T_{b+2} *and* T_{b+3}*, etc.*

Clearly, a holistic algorithm using the Tag+Level streaming is not optimal in the case of the TPQ Q_5*. The node* f_2 *remains and the configuration, where the nodes* c_2*,* e_4*, and* f_2 *are the head nodes of the lowest streams* T_{c+3}*,* T_{e+4}*, and* T_{f+5}*, will lead to a situation where all head nodes of the lowest streams are blocked. The reason is the same as in the case of TwigStack algorithm.*

PC before AD query class

The Tag+Level holistic approach is also optimal for queries where an AD edge is never before a PC edge (a PC before AD query). All query nodes having a PC between itself and parent query node can have maximally one corresponding stream. The level of this stream correspond to the level of the query node in the query tree.

2.3.3 Tag Holistic Algorithm Optimality

Due to the fact that a holistic algorithm using the Tag streaming does not involve the stream pruning phase we have to define the strong independence directly for tags in the XML document. Moreover, we have to extend slightly the original TwigStack algorithm, because the optimality can be applied only in the case that the query has at least one query match in the XML document. We have to add initial stage where we check that the query is valid.

Definition 2.3.3 *(Strongly Independent Tags) Let us have a tag* tag_{SI} *and an XML document* D*. We say that the* tag_{SI} *is strongly independent iff any node* $n \in D$ *with tag* tag_{SI} *is on the same level with any node* $n' \in D$ *with tag* tag_{SI} *such as the* $n \neq n'$*.*

An holistic algorithm using the Tag streaming (the TwigStack) is optimal if all all query nodes with a related PC edge has strongly independent tags and the query is valid. We describe valid query under Tag streaming in the next section.

Valid Query under Tag Streaming

In the previous streaming schemes we assume that the query is not processed if we have an empty set of streams in any query node after stream pruning. This basically reveals a query which is not valid. However, we do not have this phase under Tag streaming.

Even though we do not need the stream pruning in a TwigStack algorithm we can process the Tag+Level stream pruning in order to check the query validity. The Tag+Level stream pruning is not expensive and only few records per query node have to be checked. If we get an empty set in any query node, then the query is not valid.

If the query is valid and all query nodes with a related PC edge has strongly independent tags, then the holistic algorithm under tag streaming is optimal.

2.4 New Labeled Path Streaming Algorithms

Following the theoretical results achieved in Section 2.3, we propose two new holistic algorithms using the LPS: the TwigStackSorting and the TJDewey. These approaches always work with the lowest stream of the query node, and therefore Theorem 2.3.1 can be applied in their case. Each of them is a variant of the algorithm applying the Tag streaming scheme:

- The TwigStackSorting algorithm is derived from the TwigStack,
- The TJDewey is derived from the TJFast.

In Table 2.1, we show a categorization of holistic algorithms.

	Tag streaming scheme	**LPS scheme**
Element labeling scheme	TwigStack	iTwigJoin+LP TwigStackSorting
Path labeling scheme	TJFast	TJDewey

Table 2.1: Classification of the holistic algorithms

2.4.1 TwigStackSorting

The TwigStackSorting is a variant of the TwigStack algorithm for the LPS. In Table 2.1, we can observe that the TwigStackSorting is an alternative approach to the iTwigJoin+LP. The basic TwigStack algorithm remains the same with the only difference being in the *advance(q)* and *head(q)* methods.

Compared to the TwigStack algorithm, we have more than one stream per query node in the case of LPS approaches. If we want to follow Theorem 2.3.1, we have to hold streams to every query node q sorted according to the streams' head. As usual, we implement it with an array of references to the streams' head. The *advance(q)* method first shifts the head of the current stream at the next node. Therefore, the array of references has to be resorted due to the fact that it has to henceforth handle the order of streams sorted according to the streams' head. The $head(q)$ function of the query node q then simply selects the head of the lowest stream in the array. Other parts of the TwigStack remain the same.

Analysis of the TwigStackSorting algorithm

The correctness of the TwigStackSorting algorithm can be shown analogously to the TwigStack, due to the fact that they are both using the same stack mechanism. Therefore, the space, time, and I/O complexity is the same as for the TwigStack algorithm, if the algorithm is optimal. Due to this fact the TwigStackSorting is optimal for query with AD axes only.

Lemma 2.4.1 *(TwigStackSorting) If the strong independence condition is satisfied, then the TwigStackSorting algorithm is optimal.*

Lemma 2.4.1 is important for the overall performance of the algorithm. It significantly extends the number of queries for which the TwigStackSorting is optimal compared to the TwigStack algorithm.

The iTwigJoin+LP and the TwigStackSorting algorithms use the same input streams, however, their performance can be significantly different. The iTwigJoin+LP often searches the minimal value in the $soln()$ multiset and this operation has to be performed with a sequential scan within the $soln()$ multiset of streams. On the other hand, the TwigStackSorting can use the binary search algorithm during the *advance(q)* method with the logarithmic complexity.

Due to these features, the TwigStackSorting is an alternative of the iTwigJoin+LP algorithm as shown in Section 2.5, appropriate especially, when the strong independence condition is satisfied.

2.4.2 TJDewey

The second novel approach utilizing LPS is the TJDewey algorithm. This algorithm also applies the stream sorting introduced in the previous subsection. Due to the fact that this algorithm is a variant of the TJFast, we only work with leaf query nodes as well. Therefore, we keep the array only for leaf query nodes.

The majority of the TJDewey algorithm is the same with the TJFast. The TJDewey does not use the Extended Dewey; it applies the original Dewey Order labeling scheme. This labeling scheme is more appropriate for future updates compared to the Extended Dewey. Obviously, OrdPath [44] or any other path labeling scheme can be used in the TJDewey algorithm as well.

Discussion about the TJFast

The FST and Extended Dewey labeling scheme used by the TJFast is the main disadvantage with respect to updates. This approach restricts the expandability of the document schema. Future updates, which do not fit into the current schema, can lead to the relabeling of a significant part of the XML tree.

Moreover, utilizing the FST also brings the query processing overhead. We have to extract the labeled path from each Extended Dewey label and apply a string matching algorithm to determine whether the labeled path matches the pattern or not. In addition, we have to extract the *MB* set from every matched Extended Dewey label.

TJDewey algorithm

If we compare the TJFast with TJDewey, the TJDewey differentiate in the absence of *locateMatchedLabel* function used for searching the matched labeled path in the TJFast algorithm. The TJDewey also does not extract the *MB* set from every matched label, but the *MB* sets are created before the first phase of the algorithm.

Lemma 2.4.2 *(Node matching) If we apply the stream pruning, every labeled path* $lp \in PRU_q$ *in every query node* q *matches its path query pattern* p_q*.*

Proof. By Lemma 2.1.1, every $lp \in PRU_q$ has to have at least one matching labeled path lp' of class q_{parent}. We can apply this lemma recursively and get matching labeled paths for every query node in p_q, thus we are done. □

Using Lemma 2.4.2, we can omit the *locateMatchedLabel* procedure used in the TJFast algorithm due to the fact that every node matches its query pattern in the

TJDewey algorithm. Similarly to other approaches, the TJDewey works in two phases. We depict the TJDewey in Algorithms 5 and 4. In the first phase, solutions, possibly contributing to a query match, are found and stored in output arrays if the query is not optimal. The second phase can be performed efficiently, only when the solutions in output arrays are sorted. We need to block some solutions to achieve the sorting. We refer to [37] for details how to achieve this without an explicit sorting algorithm.

Algorithm 5: TJDewey algorithm

```
1 while ¬end do
2     q = getNext (root);
3     outputSolutions(q);
4     advance(T_q^min);
5 end
6 mergePhase();
```

function: outputSolutions(Query node q)

1 Return solutions of `head(`T_q^{min}`)` related to p_q such that the nodes matching the p_q exist in the ancestors' stacks.

MB set

Definition 2.4.1 *(*MB *Set) Let us have a node n of class q, a path query pattern p_q, and a query node $q_i \in p_q$. Let p_q be a vector $(q_1, \ldots, q_{m-1}, q_m)$, where the q_m is the q query node. The* MB *set is defined recursively as follows:*

$$\mathrm{MB}(n, q_i) = \begin{cases} \{n\} & \text{if } i = m, \\ \{\ n_i : \exists n_{i+1} \in \mathrm{MB}(n, q_{i+1}) & \\ \quad \text{such that the } n_i & \\ \quad \text{satisfies the structural} & \text{otherwise.} \\ \quad \text{relationship } \langle q_i, q_{i+1} \rangle & \\ \quad \text{with } n_{i+1}\ \} & \end{cases}$$

Example 2.4.1 *(*MB *Set) Let us have a node n with the $(1, 2, 3, 4, 5)$ Extended dewey label and the corresponding labeled path* `a/b/a/b/c/d` *(the FST is not important here). We want to know the* MB *set for every query node from $b//c/d$ path query pattern. Obviously,* $\mathrm{MB}(n, d) = \{(1, 2, 3, 4, 5)\}$. *Furthermore,* $\mathrm{MB}(n, c) = \{(1, 2, 3, 4)\}$. *For the query node b,* $\mathrm{MB}(n, b) = \{(1), (1, 2, 3)\}$, *due to the fact that both nodes satisfy the AD relationship with the $(1, 2, 3, 4)$ node.*

In the case of the TJFast, we use an algorithm for pattern matching. This algorithm also builds the *MB* set if the label matches its query pattern. The TJFast algorithm uses the *MB* set for a retrieval of label's ancestors. The *MB* set have to be built for every node n of class q matching its path query pattern p_q.

The situation is simpler in the case of the TJDewey. We have all labeled paths in advance and we can extract the *MB* sets from them. We do not have to extract the *MB* set from every node label as proposed in the case of the TJFast algorithm. Clearly, nodes of the *MB* set can be stored as numbers. The number indicates the length of the label. When we want to get the node $n_i \in MB(n, q)$, we use this length to extract the n_i from n.

Whole labeled path does not have to be accessible even in the case of LPS. When working with labeled paths, we usually represent the relationship by the $soln()$ multiset. Every labeled path $lp \in PRU_q$ is represented by a stream *id* (pointer to the start position of the stream) and the $soln()$ multiset for each child query node. This representation allows us to store it easily in the main memory, even when we have a high number of long labeled paths.

Algorithm 6 shows that the *MB* set for each labeled path can be extracted from a simple structure, where we keep the labeled path length and reference to labeled paths in the $soln()$ multiset. The *MB* set is empty for every labeled path at the beginning of the algorithm. Let us define the $MB(n, q).add(n_i.lpLength)$ function inserting a prefix length of an $n_i \in MB(n, q)$ into the $MB(n, q)$. We use the *prefixLen* simple array storing the labeled path length for each inner query node. We build the MB set for leaf query nodes since the TJDewey as well as the TJFast work with streams corresponding to leaf query nodes.

Analysis of the TJDewey algorithm

Output nodes in the result set are determined only by nodes in the branching nodes' stacks. This part of the TJFast algorithm remains the same in the TJDewey. Due to this fact, the TJDewey is optimal for queries with AD axes between branching nodes and their descendants and the correctness of the TJDewey algorithm is the same as the correctness of the TJFast algorithm (see [36]). The I/O and space complexity of the algorithm is also the same with the TJFast when the algorithm is optimal.

Moreover, the TJDewey is one of algorithms optimal when the strong independence condition is satisfied. The proof of Theorem 2.3.1 can be easily extended for the TJDewey since the TJDewey uses pruned labeled paths and always works with the lowest stream of a query node.

Algorithm 6: MBEnumeration(Query node q, labeled path lp)

```
1  if isLeaf(q) then
2      foreach q_i ∈ path query pattern p_q do
3          if prefixLen_{q_i} is not in MB(lp, q_i) then
4              MB(lp, q_i).add(prefixLen_{q_i});
5          end
6      end
7  else
8      prefixLen_q = lp.lpLenght;
9      foreach q_i ∈ childs(q) do
10         foreach lp_k ∈ slp(lp,q_i) do
11             MBEnumeration(q_i,lp_k);
12         end
13     end
14 end
```

2.5 Experimental Results

In our experiments[2], we test all previously described top-down approaches. These approaches have been implemented in C++. We use two XML collections: TreeBank [58] and XMARK [53] with factor 10.

Collection name	Node count	No. of labeled paths	No. of element types	Maximal depth
TreeBank	2,437,666	338,766	251	36
XMARK	20,532,805	548	77	16

Table 2.2: A Characteristic of XML collections

In Table 2.2, we observe a statistic of XML collections used in our experiments. The TreeBank collection includes many different labeled paths and the average depth of the XML document is quite high. It also contains a lot of recursive labeled paths. On the other hand, XMARK collection with factor 10 includes 512 labeled paths, where only a few of them are recursive.

Table 2.3 shows index sizes for all approaches. We use the Fibonacci encoding [66] to store the variable length labels: Extended Dewey and Dewey Order labels in this

[2]The experiments were executed on an Intel Pentium 4 1.66Ghz, 2.0 MB L2 cache; 2GB 667MHz DDR2 SDRAM; Windows XP.

Collection name	**iTwigStack + Tag [MB]**	**iTwigStack + LP [MB]**	**TJFast [MB]**	**TJDewey [MB]**
TreeBank	46.6	53.2	44.8	42.2
XMARK	393	393	291	280

Table 2.3: Index size

Name	**Query**
TB1	`//PP//NP[.//VP[.//VBG and .//SBAR//NNPS]]//_COMMA_`
TB2	`//S/VP//PP[./NP/VBN]/IN`
TB3	`//SINV//NP[./PP//JJR and .//S]//NN`
TB4	`//SINV[./ADJP[.//RB and .//JJ]]/ADVP/S[./NP]//VP`
TB5	`//_QUOTES_//S[.//VP/SBAR[./_NONE_ and .//SQ/MD and ./S/NP]] //_COMMA_`
TB6	`//EMPTY/S//NP[.//NNP and ./SBAR[.//DT and ./WHNP/PP//NN]]/_COMMA_`
XM1	`//asia/item[./description/parlist/listitem//text and ./mailbox/mail//emph]/name`
XM2	`//person[.//profile[./gender and ./business and ./gender] and .//address]//emailaddress`
XM3	`//open_auctions/open_auction[./annotation/description//text [.//bold/keyword and .//emph] and .//privacy]//reserve`
XM4	`//site//open_auction[.//bold/keyword and .//listitem/text]//reserve`
XM5	`//asia/item[.//listitem/text]//name`
XM6	`//item[.//listitem/text[.//bold/keyword and .//emph]]//name`

Table 2.4: Queries used in experiments

case. We can observe that the size of indices is almost the same in the case of TreeBank collection. Indices for the path-labeling schemes are smaller in the case of XMARK collection since the average depth is small.

Table 2.4 shows queries used in our experiments. Table 2.5 then presents query details and it involves special attributes depicting the query optimality. First, we select one query with only AD edges (TB1): column A in Table 2.5. Second, we select queries including strongly independent labeled paths in inner query nodes (TB4, TB5, XM1, XM2, XM3): column B in Table 2.5. Third, we select queries satisfying the query class for which the iTwigStack+LP is optimal as proved in [9] (TB1, TB2, TB3, XM1, XM4, XM5): column C in Table 2.5. We also select two queries which are not optimal for any approach (TB6, XM6). There are also queries with rather big number of labeled paths (TB1, TB2, TB3 TB6) and queries, where the number of labeled paths is very small (TB4, XM1, XM2, XM3, XM4, XM6).

Query	**Lp count after pruning**	**Stream pruning time [s]**	**Result size**	**A**	**B**	**C**
TB1	801	0.46	8	Yes	No	Yes
TB2	1,298	0.7	426	No	No	Yes
TB3	429	0.81	8	No	No	Yes
TB4	22	0.6	1	No	Yes	No
TB5	124	0.52	28	No	Yes	No
TB6	2,755	0.65	11	No	No	No
XM1	5	0	1,852	No	Yes	Yes
XM2	7	0	31,864	No	Yes	No
XM3	13	0	1,162	No	Yes	No
XM4	7	0	2,831	No	No	Yes
XM5	23	0	5,792	No	No	Yes
XM6	7	0	5,642	No	No	No

Table 2.5: Number of labeled paths before and after pruning

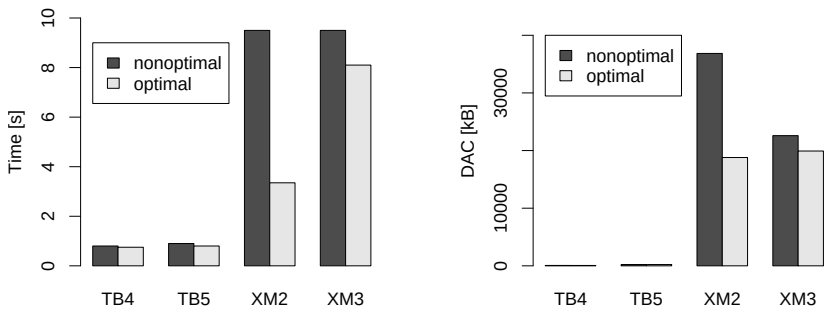
The algorithm optimality for our queries can be also expressed using columns in Table 2.5:

- TwigStack, TJFast – column A,
- TwigStackSorting, TJDewey – column A, B,
- iTwigJoin+LP – column A, B, C.

Obviously, the iTwigJoin+LP is optimal for B due to the strong independence condition proposed in this paper.

We compare all depicted approaches to the twig pattern processing in terms of *processing time*, *solution stored*, and *disk access cost*:

- *Processing time* equals to the whole time spent on twig pattern processing. This time includes the stream pruning (in the case of LPS) and both phases of holistic searching. We process every query three times and compute the average processing time. Let us note that the processing time was always very similar. Components of this time are shown in Table 2.8 and Table 2.9. We depict the time of the main memory run and the time of I/O operations.
- *Solution stored* is the number of solutions stored in the output arrays. In the case of an optimal query, no solution is stored.



(a) (b)

Figure 2.7: iTwigStack+LP optimality improvement (a) Time (b) DAC

- *Disk Access Cost (DAC)* equals to the number of disk accesses during the query processing. This is the parameter indicating the number of nodes read from the secondary storage.

Every query in the following experiments was performed separately with a cold cache. Let us also note that the cache of OS was turned off.

2.5.1 Optimality Test

We first do not apply our optimality condition and process the queries with blocking and merge phase. Afterward, we employ our strong independence condition and compare the performances of the iTwigJoin+LP approach in both cases.

The iTwigJoin+LP algorithm is not optimal for queries TB4, TB5, XM2, and XM3 according to the classification presented in [9] since they include more than one branching node. However, we show that algorithm iTwigJoin+LP is optimal for these queries because the strong independence condition is satisfied. Here we examine influence of our novel optimality condition.

In Figure 2.7, we observe how the strong independence condition affects the query processing. The most significant improvement is visible in the case of the XM2 query, due to the fact that the query result size is big and we have to block all solutions during the whole query processing.

2.5.2 Query Performance

We compare the overall performance of different approaches to the TPQ processing. We compare the TwigStack, iTwigJoin+LP, and TJFast with our TwigStackSorting

and TJDewey. In Table 2.8 and Table 2.9, we depict components of the processing time for all approaches.

We discuss different aspects of our results, because none of those algorithms is superior to the others in all queries for all measured parameters.

LPS's Random Access

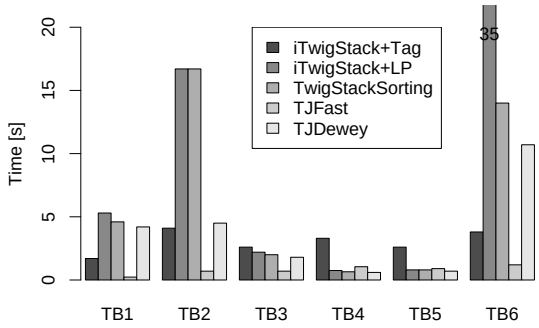
First, we consider the streaming scheme utilized and the number of labeled paths in tested XML collections. We can observe that the approaches using the Tag streaming scheme are usually more efficient if there are many labeled paths. In the case of the Tag streaming scheme, we have long input streams and the sequential search can be very fast. On the other hand, the LPS approaches have to randomly read many short streams. We can observe this issue in Table 2.8 for the queries TB1, TB2, TB3, and TB6. The TwigStack has lower I/O time than LPS approaches, but the DAC is significantly higher for these queries. For example, the DAC is 15× higher in the case of the TB3 query (see Figure 2.6(c)), however, the time is almost the same when the TwigStack and TwigStackSorting are compared. This issue would be minimized if we used a warm cache or disk array [4].

The influence of random access is not mentioned in [9], however, our results show that the random access can stand for a significant problem in queries with a high number of labeled paths.

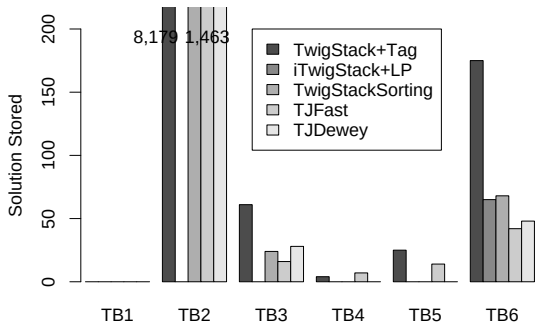
On the other hand, LPS approaches are more efficient in the case of all XMARK queries and queries TB4 and TB5. A low number of labeled paths per query node and the optimality for a larger class of queries lead to significantly better results than in the case of TJFast and TwigStack algorithms.

A Comparison of iTwigJoin+LP and TwigStackSorting

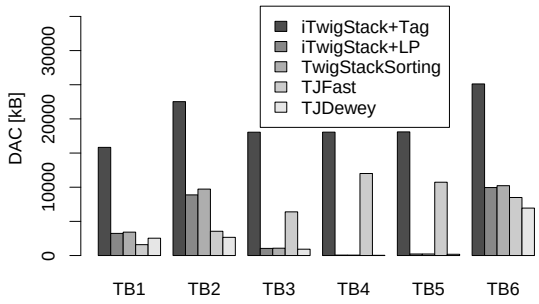
Detailed profiling of the main memory run time reveal that the stream sorting is the major part of this time in both approaches. We can observe that the influence of the stream sorting is minimized in the case of the TwigStackSorting algorithm. For example, the TwigStackSorting main memory run is in two orders of magnitude faster in the case of TB6 query (see Table 2.8). Due to this fact the TwigStackSorting is usually more efficient than the iTwigJoin+LP.



(a) Processing time

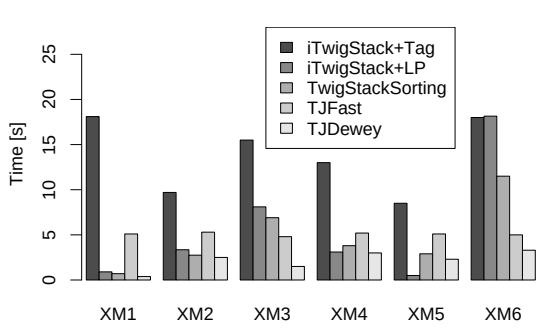


(b) Solution stored

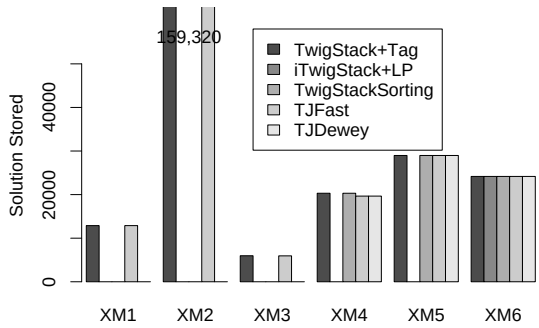


(c) DAC

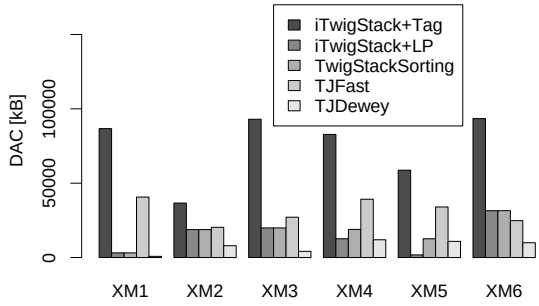
Table 2.6: Experiment results for a TreeBank data set.



(a) Processing time



(b) Solution stored



(c) DAC

Table 2.7: Experiment results for a XMARK data set.

Component	Approach	TB1	TB2	TB3	TB4	TB5	TB6
Main memory run [s]	**TwigStack**	0.16	0.28	0.37	0.83	0.67	0.63
	iTwigJoin+LP	0.37	4.1	0.19	0.01	0.01	15.61
	TwigStackSorting	0.03	0.23	0.015	0.01	0.01	0.25
	TJFast	0.08	0.25	0.26	0.45	0.45	0.58
	TJDewey	0.02	0.09	0.01	0.1	0.1	0.17
I/O operations [s]	**TwigStack**	1.2	3.7	2	2.2	1.5	2.6
	iTwigJoin+LP	4.5	11.5	1.2	0.06	0.25	13.3
	TwigStackSorting	4.5	15.4	1.5	0.06	0.25	13.5
	TJFast	0.15	0.45	0.4	0.55	0.45	0.53
	TJDewey	4.1	4.2	1.5	0.2	0.3	10.5

Table 2.8: Components of the query processing time for the TreeBank collection

Component	Approach	XM1	XM2	XM3	XM4	XM5	XM6
Main memory run [s]	**TwigStack**	2.88	1.22	1.54	1.9	1.33	1.88
	iTwigJoin+LP	0.14	1.3	1	0.58	0.08	2.36
	TwigStackSorting	0.1	0.62	0.4	0.45	0.17	0.62
	TJFast	1.7	1.75	1.6	1.5	1.1	1.5
	TJDewey	1.2	1	0.5	0.58	0.26	0.8
I/O operations [s]	**TwigStack**	14	8.5	13	10.1	5.5	16
	iTwigJoin+LP	0.75	1.8	6.5	2.2	0.3	13
	TwigStackSorting	0.75	1.8	6.5	3.8	2.4	12
	TJFast	3.2	3.5	3.1	3.7	3.6	3.3
	TJDewey	0.3	1.5	1	2.5	1.9	2.3

Table 2.9: Components of the query processing time for the XMARK collection

Labeling Scheme

The labeling scheme used by an algorithm can be another aspect. It may be observed that approaches utilizing a path labeling scheme outperform those using the same streaming scheme and the element labeling scheme. For example, the TJFast outperform the TwigStack in all queries. The TJDewey also outperforms the iTwigJoin+LP and TwigStackSorting. The XM5 query represents the only exception since the optimality of the iTwigJoin+LP algorithm for this query leads to the best results.

Since approaches applying the path labeling scheme work with leaf query nodes, a lower number of disk accesses is processed in this case (see Figure 2.6(c) and Figure 2.7(c)). In other words, the lower number of streams is retrieved.

Approaches with Path Labeling Scheme

If we compare the TJFast and TJDewey, the TJDewey is facing the same issue with the random access to the secondary storage similarly to the other LPS approaches. The time of sorting is not significant here, the TJDewey outperforms the TJFast in the main memory run even with the stream sorting. That is caused by the TJFast path pattern matching over each node label.

In Figure 2.6(b) and Figure 2.7(b), we can observe that the number of solutions stored is usually very similar for both approaches (if the TJDewey is not optimal). However, the number of DAC is considerably lower in the case of the TJDewey. That is due to the fact that the nodes matching their path query pattern in the TJFast algorithm are almost the same nodes retrieved by the TJDewey without node matching. It means that the TJFast searches possibly large streams for particular nodes, which are directly retrieved by the TJDewey. The TJFast sometimes stores some solutions which are not retrieved by the TJDewey.

Chapter 3

Stream pruning

The XML tree of the XML document is much larger than the corresponding DataGuide. Authors often consider DataGuide as a small tree. However, the DataGuide trees can be massive in the case of real XML documents or database can contain many XML documents and we store DataGuide for every one of them. Consequently, a trivial DataGuide search is time and memory consuming. In this chapter, we introduce efficient stream pruning methods for a large and complex DataGuide trees [B2].

The separation of the labeled path search from the XML node search saves some effort on a retrieval of a useless XML nodes that do not contribute to the query result at all. This advantage of the LPS is described in Chapter 2.

We adopt holistic methods, TwigStack [5] and Twig^{2}Stack [8], for the purpose of the stream pruning. Compared to the XML tree is the DataGuide still small and the DataGuide can remain the same even though that the corresponding XML document is growing. Due to this fact we use also the Twig^{2}Stack algorithm which can load the whole DataGuide into main memory in a worst-case.

We introduce a enumeration algorithm using single scan through the Twig^{2}Stack hierarchical stacks producing a duplicate-free result of a DataGuide search. We introduce a novel combination of the top-down TwigStack algorithm with the bottom-up Twig^{2}Stack to further decrease of the bottom-up approach space consumptions.

In Section 3.1, the previously published method of a DataGuide search is described. In Section 3.2, we introduce novel methods for searching a large, complex DataGuide. In Section 3.3, experimental results show that we improve the original work by up to 3×.

3.1 The Original DataGuide Search Algorithm (Stream Pruning)

The original algorithm is described in [9] as an algorithm for the stream pruning before a particular holistic algorithm. Due to the fact that the authors propose a definition without an algorithm we depict the stream pruning technique in Algorithm 7 and 8.

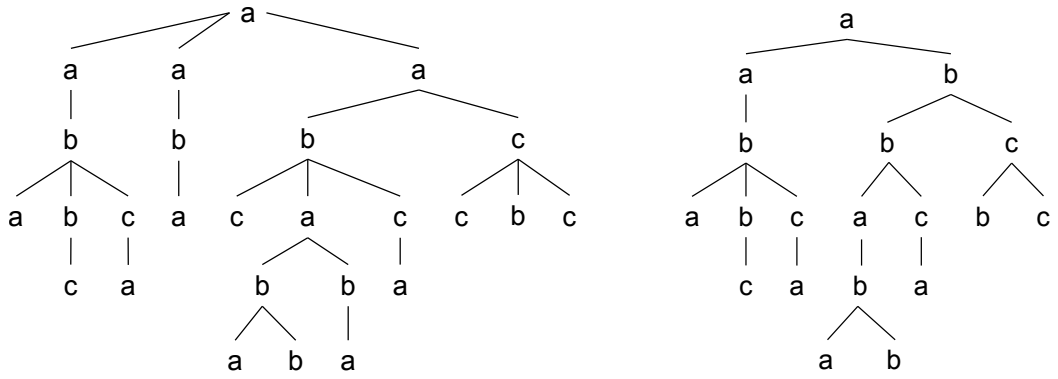


Figure 3.1: (a) Sample of the XML tree (b) Corresponding DataGuide

Algorithm 7: Stream pruning of a query pattern

input : StreamPruning(TwigPattern Q)
output: pruned labeled paths

```
1 U ← {};
2 for ∀lp of class q_root do
3     U ← U ⋃ StreamPruning(q_root,lp);
4 end
5 return U;
```

At beginning of the stream pruning we have the U_q set for every $q \in Q$ which includes all labeled paths of class q. U_q is retrieved for each leaf of a twig pattern (see Algorithm 8, Lines 3, 7, and 8). If a q pattern node is the inner node, we must process all labeled paths of q's children (Lines 5–15). In Line 7, a condition for adding labeled paths into the result is depicted. If there is a child with no solution labeled path, the lp produces an empty result (Line 11).

Algorithm 8: Stream pruning of a query node

input : StreamPruning(TwigPatternNode q, LabeledPath lp)
output: a set of matched labeled paths

```
1  U ← {};
2  if q == leaf then
3      U ← lp;
4  else
5      for ∀q^c ∈ child(q) do
6          U_c ← {};
7          for ∀lp_c ∈ slp(lp, q^c) do
8              U_c ← StreamPruning(q^c,lp_c);
9          end
10         if U_c == ∅ then
11             return {};
12         else
13             U ← U ⋃ U_c;
14         end
15     end
16 end
17 return U;
```

Example 3.1.1 *(Stream Pruning Example)*

Let us consider the DataGuide in Figure 3.1(b) and the TPQ //b[/a]//c.

In the first step, we retrieve all labeled paths of leaf node class a and c, respectively.

$$U_a = \{ \{a\}, \{a/a\}, \{a/a/b/a\}, \{a/a/b/c/a\}, \{a/b/b/a\}, \{a/b/b/a/b/a\}, \{a/b/b/c/a\}\}$$

$$U_c = \{ \{a/a/b/b/c\}, \{a/a/b/c\}, \{a/b/b/c\}, \{a/b/c\}, \{a/b/c/c\}, \}$$

In the second step, we retrieve all labeled paths of inner node class b.

$$U_b = \{ \{a/a/b\}, \{a/a/b/b\}, \{a/b/b\}, \{a/b/b/a/b\}, \{a/b/b/a/b/b\}, \{a/b/c/b\}\}$$

In the third step, we select every labeled path with solution in both U_a and U_c sets.

In other words, an lp is in the result if $\exists lp_a \in U_a : slp(lp, a) = lp_a \wedge \exists lp_c \in U_c : slp(lp, c) = lp_c$. *Obviously, some labeled paths are pruned. For example, the* `a/a/b/b` *labeled path produces an empty set due to the fact that* `a/a/b/b/a` *is not included in the* U_a. *The result is following:*

$$\begin{aligned} U_r = \{ \quad & \{\{a/a/b\}, \{a/a/b/a\}, \{a/a/b/b/c\}, \{a/a/b/c\}\}, \\ & \{\}, \\ & \{\{a/b/b\}, \{a/b/b/a\}, \{a/b/b/c\}\}, \\ & \{\}, \\ & \{\}, \\ & \{\}, \end{aligned}$$

3.1.1 Stream Pruning Efficiency Issues

The space and time complexity is rather high. In Example 3.1.1 we can observe that many labeled paths are processed, however, only few labeled paths are in the result. This issue is accented in the case of a real collection.

Let us consider the Treebank [58] data collection, 82 MB in size. The number of nodes is 2,437,666. The maximal depth is 36. The number of element types and labeled paths is 251 and 338,766, respectively. Obviously, the DataGuide tree is rather large and complex.

Let us consider the TPQ `//PP[./NP//VBN]/IN`. We must process 5,803 labeled paths of class `VBN`; 49,857 of class `NP`; 19,405 of class `PP`; and 22,142 of class `IN`. Consequently, we process 97,207 labeled paths. After the *slp* computation for each inner query node, 7,557 labeled paths are in the result. We must handle 13× more labeled paths than necessary. Thus, time and memory consuming feature of the trivial stream pruning technique is obvious. If we prune the high number of labeled paths, this feature becomes significant.

3.2 Novel Approaches to Search Large DataGuide Trees

In the following sections, we discuss how can be existing holistic approaches used during the stream pruning. In all cases, we consider the $soln()$ multiset as an expected output of the search. We have two basic operations used by all holistic algorithms. Let us suppose that a q pattern node is an object with attribute LP – a set of labeled paths. The first operation $q.put(lp)$ pushes the new labeled path lp of class q into

the $q.LP$. The second operation $soln(lp, q_i).add(lp')$ pushes the labeled path lp' of class q_i into the $soln(lp, q_i)$ multiset, where lp is of class q and q_i is a child of the q node.

All holistic approaches can be adapted for the DataGuide labeled path search. First, we set some node ordering in the DataGuide. We can use the order in which the nodes are inserted into the DataGuide or we can order nodes according to their names. Consequently, we evaluate every node in the DataGuide using the containment labeling scheme and store labels in the inverted list. The key of the inverted list is the node name. One element (label) here stands for one labeled path and it is sometimes used in the same meaning in the following text. Every element contain the pointer to the stream of XML nodes lying on this labeled path (represented by this elements). These pointers are then stored in the *soln*.

3.2.1 TwigStack

The TwigStack algorithm was described in Section 2.2.1. The first phase of the holistic algorithm remains the same, because we work with labeled elements stored in streams. The only difference is in the result, because we want to retrieve the $soln()$ multiset from the DataGuide. We call the phase of $soln()$ retrieval as an *enumeration*.

In the second phase, TwigStack merges output paths and prunes useless paths. It was proved [5] that TwigStack is optimal only for AD queries. In cases of a query with the PC axis, the algorithm may produce useless paths. Consequently, the result must be post-processed by a merge-join algorithm.

Enumeration Algorithm

When we are enumerating the $soln()$ mutliset for an AD query we call the $q.put()$ and $soln(lp, q_i).add()$ functions while pushing an element into the query node stack. This situation is not as simple when working with a PC axis in the query. Output paths may contain many duplicate nodes and we must know when the $q.put(lp)$ function can be called.

By using stack for every query node during the merging, we are able to determine the proper condition for a $q.put(lp)$ function. If we find a query match Q_m we pop all elements from the S_q stack which are not ancestor-or-self of a node n of class q where node $n \in Q_m$. We put the labeled path lp of class q into the $q.LP$ only if $\langle q, q_{parent} \rangle$ is a PC relationship, or if $\langle q, q_{parent} \rangle$ is an AD relationship and stack

$S_{q_{parent}}$ is empty. This simple rule allow us to enumerate the $soln()$ function during the merging phase if the algorithm is not optimal.

3.2.2 Twig^{2}Stack

The Twig^{2}Stack [8] uses hierarchical stacks to store unresolved elements in the main memory. It utilizes a bottom-up query processing algorithm. Elements for the same query node are stored in a tree structure in the main memory, and in this way the Twig^{2}Stack enables result enumeration without the merge-join phase even in the case of a TPQ with a PC relationship.

The Twig^{2}Stack algorithm is based on a hierarchical stack encoding scheme capturing the AD relationship between elements in the same query node. Every query node q of a twig query Q has a corresponding hierarchical stack $HS[q]$. $HS[q]$ consists of a list of ST stack trees, where ST is an ordered tree of possibly empty S stacks. An element in an ST is the ancestor for all elements below in the same stack and it is also an ancestor for all elements in its descendant stacks. The $subST$ may be a subtree of an ST. We can easily create the hierarchical structure during reading elements from a stack in a post-order (bottom-up technique). For details see [8].

The post-order is simply generated by reading elements in pre-order using a global stack. The elements in $HS[q]$ of a q inner query node also contain pointers to matching elements of its child query nodes. The Twig^{2}Stack ensures that an element of an inner query node is never pushed to a stack if it does not have all matching elements in its child query nodes. We also store matching elements and child stack pointers together with elements for the inner query node. Function $pointer(e, child)$ returns a root of a $subST$ in an $HS[child]$ for an element e of class q, where q is a parent of the $child$ query node.

Example 3.2.1 *(Hierarchical stacks) In Figure 3.2, we see an example of hierarchical stacks generated from the DataGuide in Figure 3.1 for the TPQ* `//b[.//a]/c`*. We create hierarchical stacks for every query node and they form a set of STs within HS. For example, stacks in the $HS[a]$ create one ST tree. The pointer function $pointer(b1, a)$ returns the stack with the $a3$ element and function $pointer(b1, c)$ returns the stack with the $c2$ element.*

The hierarchical stacks are created during the scan of input streams. It is particularly important to push elements into hierarchical stacks in the post-order to certainly match all results.

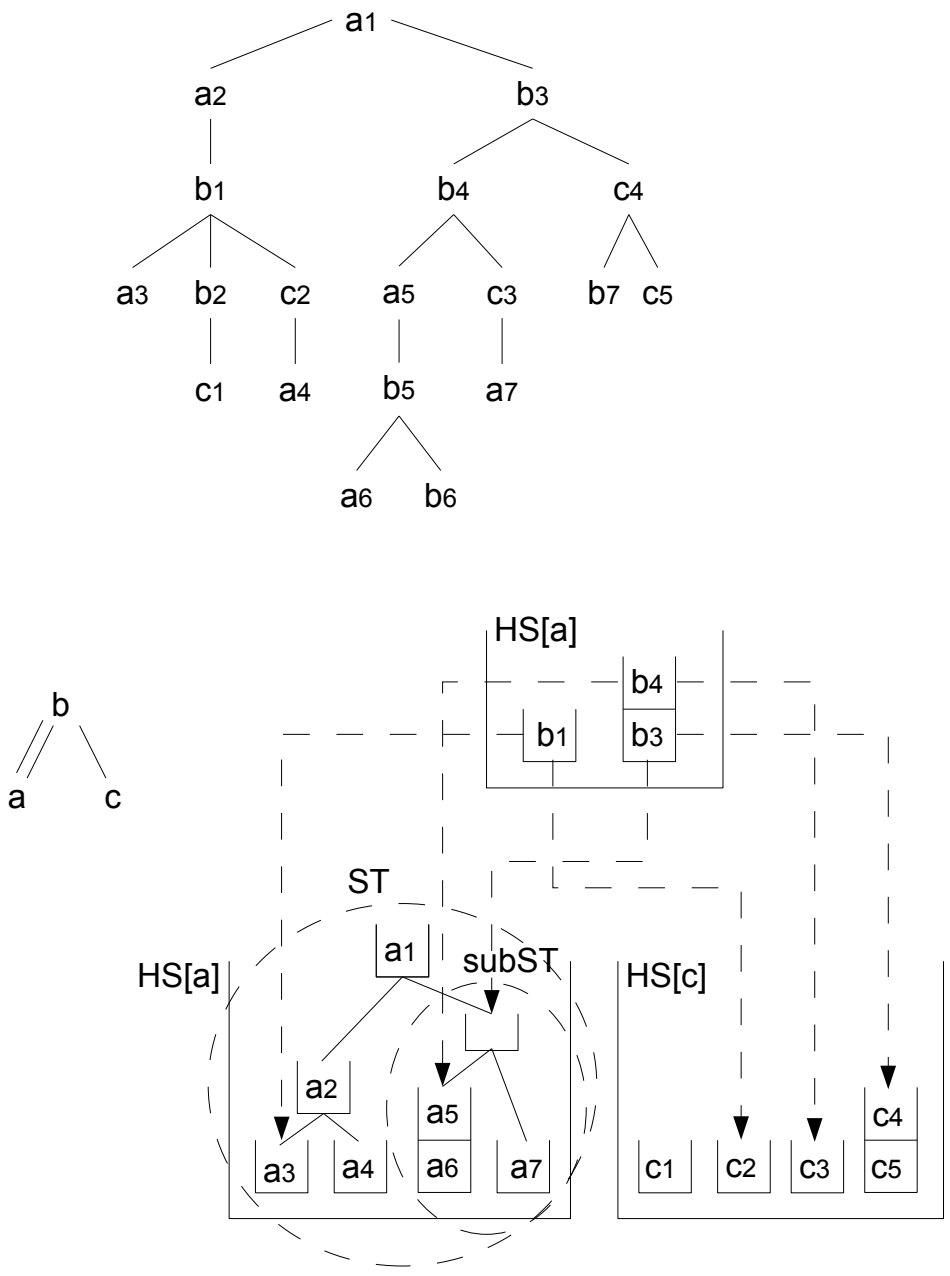
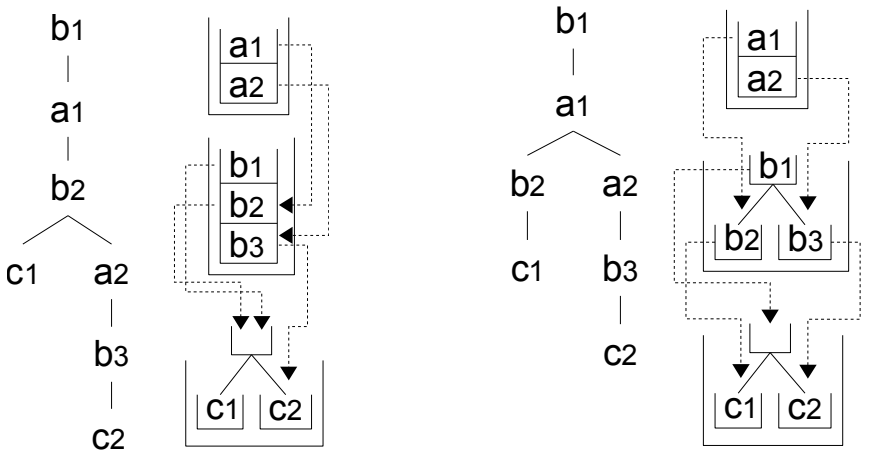


Figure 3.2: Sample of the DataGuide, twig query pattern, and its corresponding hierarchical stacks.



(a) (b)

Figure 3.3: Two sample documents with corresponding hierarchical stack for the TPQ `//a/b//c`

Enumeration Algorithm

During the enumeration we traverse the hierarchical structure from elements in the root query node. A main issue during the enumeration is to decide whether or not the actual element of class q is already in the $q.LP$ set (is duplicated). Original work [8] proposes to enumerate only the root element of an ST from $HS[a]$ when enumerating the AD relationship. However, the hierarchical stacks may contain useless nodes and that may cause problems if we follow previous recommendation as we show in Example 3.2.2. If we want to resolve element duplication in the whole query this information is not sufficient.

Example 3.2.2 *(Element Duplication) Let us have the document in Figure 3.3(a) and the hierarchical stacks for the query* `//a/b//c`*. When we enumerate the* $b2$ *element, we observe that the element* $b2$ *is not the top one in* ST*, however,* $c1$ *and* $c2$ *elements have not been enumerated yet. Only elements from the root query node can be recognized at the top of an ST stack. This is not sufficient for our issue.*

In Algorithm 9, we propose our algorithm capable of resolving the $soln()$ multiset for all relevant labeled paths during one hierarchical stack scan. During each call of the Enumerate() method we evaluate the single element e in the hierarchical stacks.

The current element e may be stored in the query node as a new labeled path (Line 2) and added into the parent *soln* multiset (Line 3).

The `false` value of the *duplicit* parameter indicates that the current element was already put into the query node and resolved previously. The proper value of this parameter is mainly driven by *top* and *axis* parameters (Lines 11 and 12). We delegate the *top* information from the root node to the descendant nodes. This information can be sufficient for a selection of the proper value of the *duplicit* parameter for child elements in many cases. However, this information is not sufficient in all cases. For this reason, we must keep the previously evaluated element of class q in the array *prev*. The values of the *prev* array are mainly set in Algorithm 10 (Lines 7–11) and in Algorithm 9 (Lines 24–26).

Example 3.2.3 *(prev Array Necessity) In Figure 3.3, we observe examples of two different documents along with the hierarchical stacks for the same query* `//a/b//c`*. When enumerating the* $b3$ *element, the value of the top variable is* `false` *and the value of the duplicit variable is* `false` *in both documents. The axes are the same as well since we enumerate the same query. The* $c2$ *element, which is the descendant of the* $b3$ *element, is duplicated in the document in Figure 3.3(a), however, this element is not duplicated in the document in Figure 3.3(b). We must compare the* $b3$ *element with the element stored in the* $prev[b]$ *array to set the value of the duplicit variable correctly. In this case, the element stored in the* $prev[b]$ *is the* $b2$ *element and this comparison reveal that the descendants (not in the same query node) of the* $b3$ *are duplicit elements.*

3.2.3 TwigStack and Twig^{2}Stack Comparison

The TwigStack algorithm is optimal for AD queries and enables us to omit a high number of elements for queries with a PC axis. However, the enumeration of blocked items can also lead to random accesses into the secondary storage. Output paths may contain duplicate elements for inner nodes as well. For example, consider the TPQ `//a[./b]/c`, where the output paths matching the query are a_1, b_1 and a_1, c_1. We observe that the a_1 node is duplicated.

The Twig^{2}Stack algorithm enables us to omit the merging phase and each element is only stored once in the hierarchical stacks. On the other hand, it may store a high number of unnecessary nodes in the hierarchical stacks and it stores many elements which would be omitted by the TwigStack algorithm.

In Section 3.2.4, we introduce a simple fusion of the TwigStack and the Twig^{2}Stack to combine advantages of both algorithms.

Algorithm 9: Enumeration algorithm

input : Enumerate(QueryNode q, Axis $axis$, Element e, bool top, bool $duplicit$, Set $parentsoln$)

```
1  if not empty(e) then
2      if not duplicit then q.put(e);
3      if isRoot(q) then parentsoln.add(e);
4  end
5  if isLeaf(q) then
6      SubST(q, axis, e, top, duplicit, parentsoln);
7  else
8      if not duplicit then
9          for ∀d ∈ child(q) do
10             ch_dup = false;
11             if not top and q → d == AD then
12                 if axis == PC then
13                     top = true;
14                     if prev[q].ancestor(e) then
15                         ch_dup = true;
16                     end
17                 else ch_dup = true;
18             end
19             Enumerate(d,q → d, pointer(e,d), top, ch_dup, q.soln(e,d))
20         end
21     end
22     SubST(q, axis,e, top, duplicit, parentsoln);
23 end
24 if axis == PC and not prev[q].ancestor(e) then
25     prev[q] = e;
26 end
```

Algorithm 10: SubST algorithm

```
1
  input : SubST(QueryNode q, Axis axis, Sot sot, bool top, bool duplicit, Set
          parentsoln)
2 if axis == AD then
3     if top and empty(e) then top ← true else top ← false;
4     for all childs c of e in the ST do
5         Enumerate(q, axis, c, top, duplicit, parentsoln);
6     end
7     for ∀d ∈ child(q) do
8         if q → d == PC and not prev[d].ancestor(pointer(e,d)) then
9             prev[d] = pointer(e,d);
10        end
11    end
12 end
```

3.2.4 Fusion of TwigStack and Twig^{2}Stack

The Twig^{2}Stack algorithm can potentially store all elements from streams in the main memory. It may also push many unnecessary elements into hierarchical stacks. This issue is presented in Figure 3.2 where elements a_1, a_2, a_4, c_1 are useless and elements a_1 and a_2 may be simply omitted during the search.

The Twig^{2}Stack algorithm proposes the early enumeration based on PathStack which helps to omit some nodes and reduces main memory consumption. A document element visited during the pre-order traversal is first processed by the PathStack algorithm.We push the an e element into $HS[E]$ only if this element is popped from the E stack and this decreases memory requirements.This solution also enables the Twig^{2}Stack to enumerate results before the algorithm ends. When the top branching node stack of the PathStack algorithm is empty, all hierarchical stacks can be enumerated and cleaned.

However, the PathStack is only applicable for a simple path query. Therefore, we want to utilize a top-down algorithm for twig query patterns like the TwigStack algorithm. The only problem we have to solve when using this approach is how to push elements into hierarchical stacks in the post-order way. This is an important issue because the TwigStack does not always push or pop elements in the pre-order or post-order way. We can not directly push elements from the TwigStack's stacks into the Twig^{2}Stack because the Twig^{2}Stack requires post-order to work properly.

We use the L_q queue for each query node to achieve post-order. When we pop an element from a TwigStack's stack we do not push the element into the hierarchical

stacks until we are sure that no other unresolved element with lower post-order is in the TwigStack's stacks. This is decided by checking the top element of the query node stack and the stream's head element. If there is a possibility of such element in the TwigStack, the element popped from S_q is first pushed into the L_q queue. The queues are checked every time we pop an element from a stack.

This novel approach results in a considerably lower number of elements in the Twig^{2}Stack hierarchical stacks. It also saves some time on a manipulation with hierarchical stacks. This is shown in our experiments.

3.3 Experiments

In our experiments[2], we test all previously depicted approaches to searching in the DataGuide. We implemented the approaches in C++. We use two XML collections, TreeBank [58], and synthetic genealogy collection[1]. We compare their time and memory consumption during query processing.

3.3.1 XML Collections and Queries

Collection name	Number of labeled paths	Number of element types	Maximal depth
TreeBank	338,766	251	36
Genealogy	155,912	45	32

Table 3.1: XML collections characteristic

In Table 3.1, we observe the statistics of XML collections used in our experiments. We choose collections with a rather high number of labeled paths where the difference between the approaches is measurable. The synthetic genealogy collection includes the 45 most common male names, and the structure is created randomly. However, the distribution of the names is not uniform. We use simple quadratic distribution. No element in the XML tree contains more than five children.

Table 3.2 put forwards queries used in our experiment. We have selected queries only containing a PC axis (TB3, G1) or an AD axis (TB6, G3), and queries containing

[2]The experiments were executed on an Intel Pentium 4 1.66Ghz, 2.0 MB L2 cache; 2GB 667MHz DDR2 SDRAM; Windows XP.

[1]`http://www.cs.vsb.cz/arg/src/Genealogy.java`

Name	Query
TB1	`//VP[./DT and ./_NONE_]//PRP_DOLLAR_`
TB2	`//S//NP[.//PP/TO and ./VP/_NONE_]/JJ`
TB3	`//S/PP[./NP/VBN]/IN`
TB4	`//S//NP[.//PP[./TO and .//VBN] and .//VP/_NONE_]/JJ`
TB5	`//SBAR[.//VBN and .//S//X//MD]`
TB6	`//VP/S[.//NP[./CD and ./POS] and ./X//JJ]/_COMMA_`
G1	`//Jacob[./Elijah and ./Ryan]/Alexander`
G2	`//Johnathan//Matthew[./Jacob and ./Noah]`
G3	`//Daniel[.//William[.//Anthony and .//Andrew] and .//Samuel]//Mason`
G4	`//David[./Nicholas and .//Joshua/Ethan]/Matthew`
G5	`//John//Elijah[./Michael and .//Jacob]`
G6	`//Tyler[./Michael//Jose and ./Jacob]//Daniel`

Table 3.2: Queries used in experiments

different number of query paths. There are also queries containing two branching nodes (TB4, TB6, G3). We also selected queries with a rather low number of result labeled paths (TB3, G4, G6) and with a high number of result labeled paths (TB1, G3). In Table 3.3, we observe the number of labeled paths for every query before and after the stream pruning.

We compare approaches to the labeled path search in terms of *processing time*, *memory consumptions*, *labeled path stored* and *disk access cost*:

- *Processing time* measures time spent on *soln* search. This time involves basic *soln* construction and stream pruning in the case of the Stream pruning algorithm. In the case of holistic approaches, this time involves the time of a holistic algorithm for a twig query matching and enumeration of a result.

- *Memory consumption* measures the peak memory requirements during the *soln* search. The Stream pruning algorithm needs to store initial sets of labeled paths and it also needs to store the whole U array in the main memory. The TwigStack only uses small cache for persistent array of blocked items and output arrays. That is the only memory requirement of the TwigStack, because the size of stacks is insignificant. On the other hand, the Twig^{2}Stack may store a high number of elements in hierarchical stacks. We measure all memory requirements consumed by these stacks such as stack pointers, element pointers, and elements themselves. The fusion approach also stores labeled paths in hierarchical stacks.

Query	Number of streams before pruning	Number of streams after pruning
TB1	50,847	2,408
TB2	132,426	1,870
TB3	107,955	50
TB4	138,229	2,098
TB5	23,593	2,248
TB6	110,278	290
G1	30,995	12
G2	33,995	12
G3	21,659	1,216
G4	25,510	7
G5	36,368	146
G6	41,287	10

Table 3.3: Number of labeled paths before and after pruning

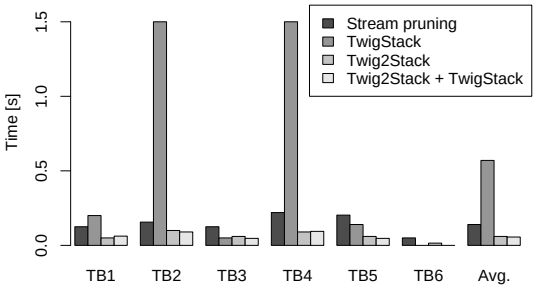
- *Labeled path stored* is the number of labeled paths to be stored for another processing. Obviously, the Stream pruning stores all labeled paths to each query node for another processing. On the other hand, holistic approaches filter some irrelevant labeled paths before they are pushed into the stacks and enumerated. We count only the number of labeled paths pushed into hierarchical stacks in the case of fusion approach because only these labeled paths are enumerated.

- *Disk Access Cost (DAC)* - we also measure the number of disk accesses during the query processing. Only the TwigStack works with secondary storage arrays during the query evaluation. All the other approaches only read the elements from the input streams. The remaining work is then processed in the main memory.

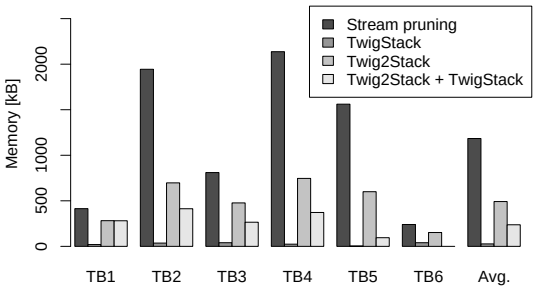
3.3.2 Experimental Results

In this experiment, we measure results for queries in Table 3.2. To obtain the result times as precise as possible each experiment was processed five times and we choose the result to occur the most frequently. We must note, that the results were very similar. The result graphs can be seen in Figure 3.3.2 and in Figure 3.3.2.

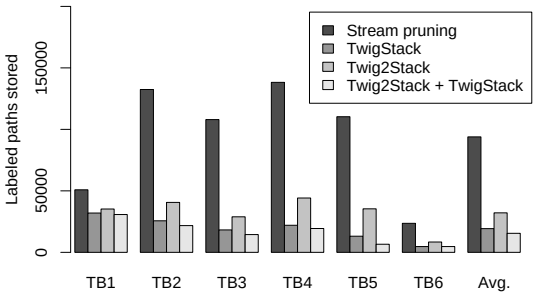
The memory requirements and processing time in the case of the Stream pruning



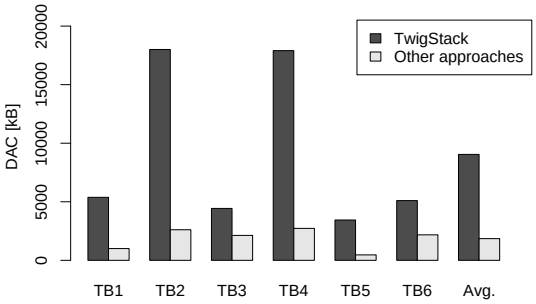
(a) Processing time



(b) Memory consumptions

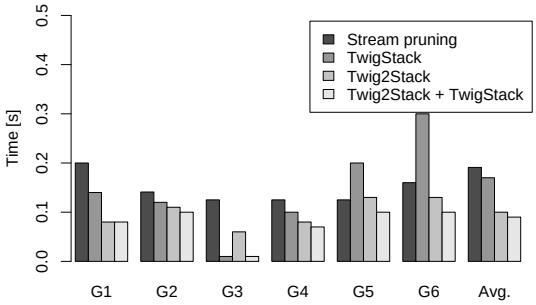


(c) Labeled path stored

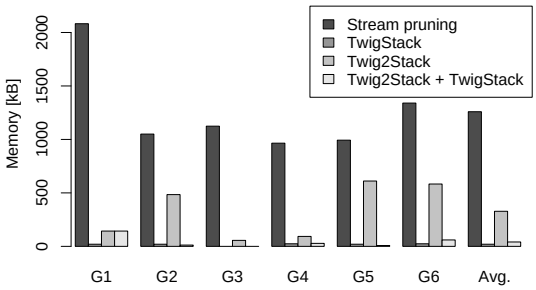


(d) DAC

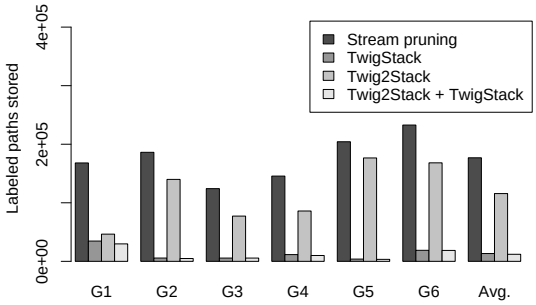
Table 3.4: Experiment results for a Tree-Bank data set.



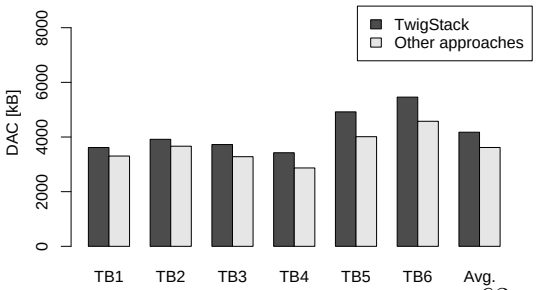
(a) Processing time



(b) Memory consumptions



(c) Labeled path stored



(d) DAC

Table 3.5: Experiment results for a XMARK data set.

is very dependent on a size of input streams. The performance of the TwigStack algorithm is the best in the case of AD query (TB6, G3), when a merge-join phase and element blocking can be skipped. However, in few cases (TB2 and TB4) the TwigStack performs significantly worse in terms of time and DAC than the other approaches. The TwigStack blocks high number of solution paths during query processing and reading these solution paths leads to a high number of random disk accesses. This support our statement in Section 3.2.1 about the problems with blocking array.

The Twig^{2}Stack algorithm works more efficiently because it stores the elements in a main memory. Utilization of the early enumeration algorithm keeps the memory consumption low, however, the number of labeled paths pushed into the hierarchical stacks is significantly higher than in the case of the TwigStack algorithm. The advantages of our fusion of the TwigStack and the Twig^{2}Stack are obvious. It helps to decrease the number of stored labeled paths in the hierarchical stacks and decreases the memory consumption. It also enables us to enumerate results directly in the case of AD query without using the hierarchical stacks at all. It uses the hierarchical stacks instead of blocking array, and therefore there is no secondary random access issue.

We observe that the scalability of introduced algorithms is high: the improvement of processing time is up to 3× higher; memory consumption and labeled path stored are both improved by up to 4×. Moreover, this issue may be arisen in the case of searching more DataGuide trees.

Chapter 4

XML Document and an Holistic Algorithm Optimality

Designing a proper structure of an XML document can be a similar procedure to a relational database schema design. There can be found works giving advices about the design of a proper schema of an XML document [52, 2, 15, 6, 14]. The goals of such XML schema design recommendations can be a future extensibility of the schema or a compatibility of cooperating systems [52]. Another challenging topics in an XML schema design are, similarly to relational databases, redundancy avoidance and preservation of a data integrity [2, 15, 6, 14].

By a term 'XML schema' we mean any metadata language like an XML Schema [64] or DTD [59] which is capable of specifying the XML document schema.

In this chapter, we propose a different view on a proper XML schema design. We propose basic definitions for an XML document an its corresponding XML schema, where we can guarantee a holistic algorithm optimality. This means that we can guarantee the worst case time, I/O, and space holistic algorithm complexity for any TPQ considered in the XML schema design.

These definitions can help to understand the types of XML documents and TPQs, where the holistic approaches are particularly suitable. It also reveals the weak points of the holistic methods in the context of the XML document, which allows users to select appropriate indexing method.

In the following sections we reverse the strong independence condition and we analyze what have to be satisfied for an XML document in order to satisfy the strong independence condition for any query or at least for a set of queries.

In Section 4.1 we introduce basic definitions bounded with a strong independence of an XML document or an XML schema. In Section 4.2 we define the strongly

independent XML document under the Tag and Tag+Level streaming scheme. In Section 4.3 we describe the strong independence of an XML document under the LPS scheme and in Section 4.4 we generalize this concept for a $SI \subseteq TGS$.

4.1 Basic Definitions

The strong independence condition does not have to be satisfied only after the stream pruning. We can find XML documents, where the structure is non-recursive and we can draw some conclusion about the XML document with respect to LPS holistic algorithm optimality.

We start with few the definitions for the XML document D with the corresponding XML tree X. The definitions are the following:

- **Valid TPQ** - We say that that TPQ Q is a valid query for the XML document D if any query node $q \in Q$ has at least one stream in PRU_q under any streaming scheme (Tag, Tag+Level, LPS).

- **Single level tag** - The tag $tag_n \in D$ is a single level tag, if the node is always on the same level in the XML tree.

- **Recursive tag** - The tag $tag_n \in D$ is called recursive if there exists the inner node $n \in X$ with tag tag_n having the node $n' \in X$ with tag_n as an ancestor.

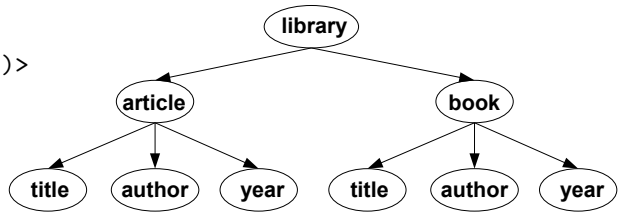
- **Recursive labeled path** - The labeled path $lp \in D$ of class tag_{lp} is recursive if the tag_{lp} is the recursive tag.

Definition 4.1.1 *(Graph of the XML document schema) Let us have the XML schema S. The graph $G(N, E)$ of the XML schema S is an oriented graph, where every node $n \in N$ corresponds to one element or attribute in the XML schema S and edge e points from a node n_1 to node n_2 if the nodes can have the PC relationship according to the XML schema S.*

Example 4.1.1 *(Graph of the XML schema) In Figure 4.1(b) we can see the graph of the XML schema in Figure 4.1(a). Other symbols like question mark or plus are omitted in the graph.*

```
<!ELEMENT library (article?|book?)+>
<!ELEMENT article (year,author?,title)>
<!ELEMENT book (year,author,title)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
```

(a)



(b)

Figure 4.1: (a) Schema of the XML document in DTD (b) Graph of the schema

4.2 Strongly Independent XML Document under Tag and Tag+Level Streaming

First we define the XML document optimality under Tag and Tag+Level streaming scheme.

Definition 4.2.1 *(Strongly independent XML document under the Tag and Tag+level streaming scheme) Let us have an XML document D. We say that the XML document D is strongly independent under the Tag and Tag+Level streaming scheme if all tags in the TGS_D are single level tags.*

It can be shown that the strongly independent XML document under the Tag and Tag+Level streaming scheme cause that any valid TPQ satisfies the strong independence condition under the Tag and Tag+Level streaming scheme, respectively.

Basically we have only one stream per tag even under Tag+Level streaming scheme. Therefore, we only check the validity of a query. None of the query nodes can contain more than one stream, which is the condition for strong independence under the Tag+Level streaming.

Definition 4.2.2 *(Strongly Independent XML schema under the Tag and Tag+Level Streaming Scheme) Let us consider an XML schema S and a graph $G(N, E)$ of the XML schema S. We say that the XML schema S is strongly independent under the Tag+Level and Tag streaming scheme if the graph $G(N, E)$ is a tree, where the root of the tree is the root element in the XML schema S and the nodes $n \in N$ with the same tag have the same distance from the root node.*

Example 4.2.1 *(Strongly Independent XML Document and schema under the Tag and Tag+Level Streaming)*

In Figure 4.2 we can see the example of the strongly independent XML document under Tag streaming and Tag+Level streaming. Graph of its corresponding XML schema is in Figure 4.3, where the root node is the library tag. We can observe that every tag which occurs more than once in the XML schema has the same distance from the root node, and therefore the XML schema is strongly independent under the Tag and Tag+Level streaming scheme.

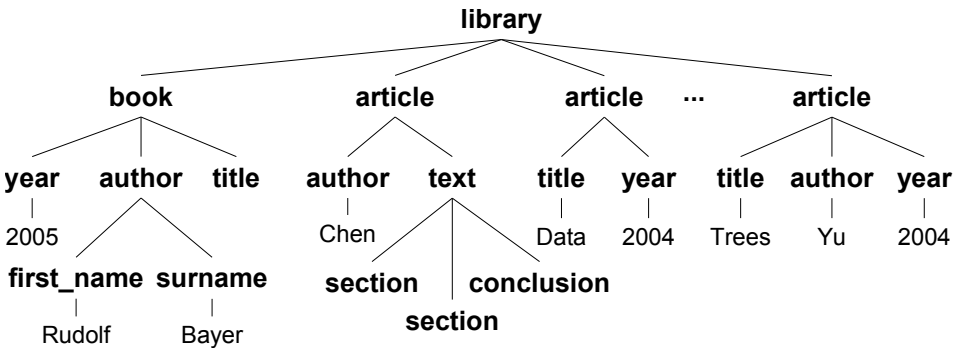


Figure 4.2: Strongly independent XML document under the Tag and Tag+level streaming scheme

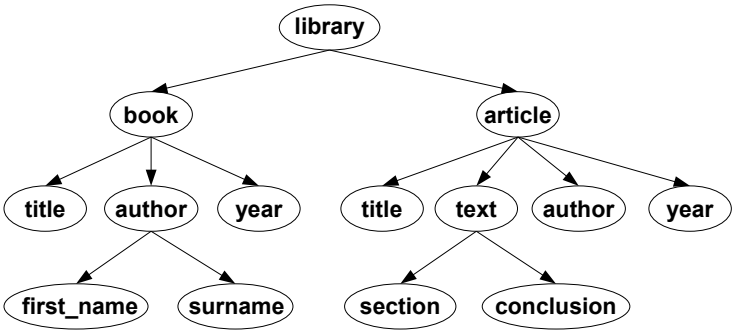


Figure 4.3: Graph of the strongly independent XML schema under the Tag and Tag+level streaming scheme

Any valid TPQ will have only one stream per query node, and therefore it will be strongly independent under the Tag+Level and Tag streaming.

4.3 Strongly Independent XML Document under LPS Scheme

Now we consider only the LPS scheme and the strong independence under LPS. We do not include the 'under the LPS scheme' term in the definitions for the sake of abbreviation.

Definition 4.3.1 *(Strongly independent XML document) Let us have an XML document D. We say that the XML document D is strongly independent iff every $tag \in TGS_D$ is non-recursive.*

Theorem 4.3.1 *Let D be a strongly independent XML document. Then any valid TPQ Q for the XML document D satisfies the strong independence condition.*

Proof. Note that if every tag tag_n is non-recursive, then also every labeled path lp of class tag_n is non-recursive. This means that we can define a strongly independent XML document as an XML document, where every labeled path is non-recursive. If every labeled path is non-recursive, then every labeled path $lp \in PRU_q$, for any valid TPQ Q and $q \in checkingnode(Q)$, has to be strongly independent due to its non-recursive character, and therefore the strong independence condition has to be satisfied for any valid TPQ Q for the document D and we are done. □

Example 4.3.1 *(Strongly independent XML document) The XML document in Figure 4.4 is the example of the strongly independent XML document since all nodes in this document are non-recursive. Note that the definition for the recursive tag is specified only for inner nodes and thus the node section is not a recursive node. Let us now explain why the recursive character of labeled path library/book/section/text/section does not represent a problem here. If we have the TPQ, where the section query node is an inner query node, then the library/book/section/text/section labeled path will be certainly pruned during the stream pruning. If the section query node is the leaf in the query tree Q, then the recursiveness of the labeled paths is not a problem because the checkingnodes(Q) set contains only inner query nodes.*

It can be observed that the XML document in Figure 4.4 is not strongly independent under the Tag+Level streaming. Main problem would be queries containing PC edge between query nodes book and author like the TPQ `//book[./author]/title`*. Every query node has two streams in this TPQ under Tag+Level streaming.*

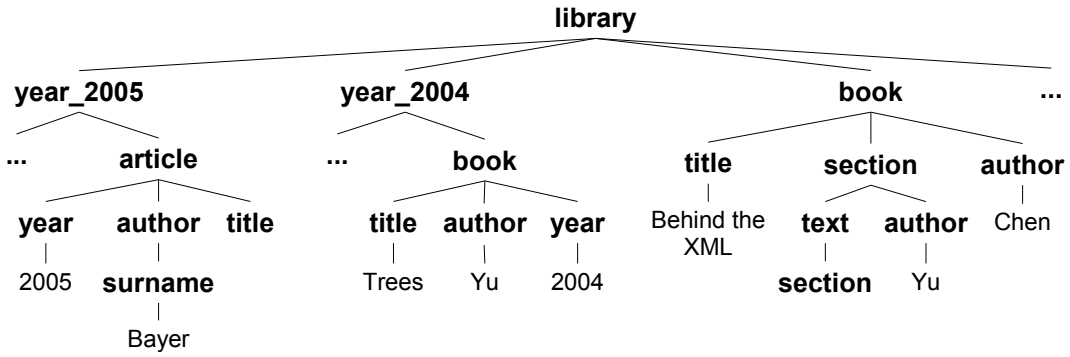


Figure 4.4: XML document only with strongly independent labeled paths

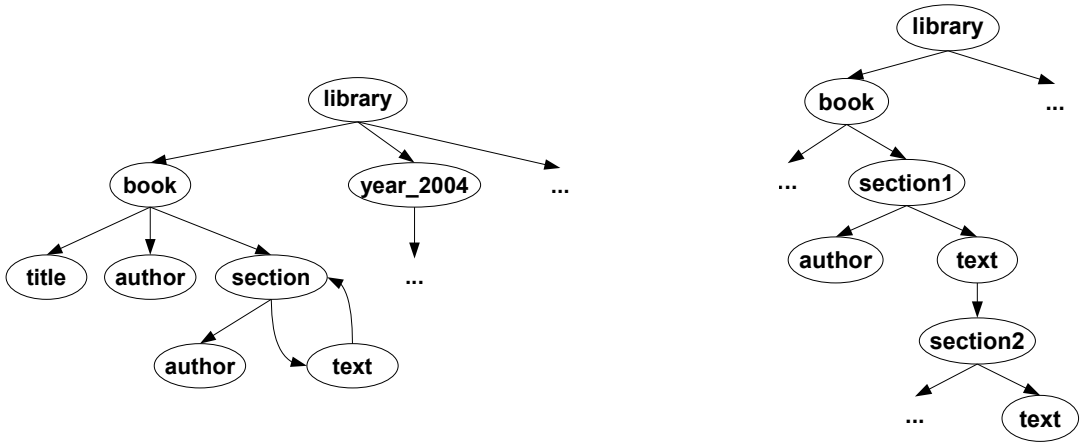
The strong independence is connected to an XML document, but it is possible to define it also for an XML schema.

Definition 4.3.2 *(Strongly independent XML document schema) Let us have an XML schema S and a graph $G(N, E)$ of the XML schema S. We say that the XML schema S is strongly independent if the graph $G(N, E)$ is a rooted tree, where the root of the tree is the root element in the XML schema S and any inner node $n \in N$ with tag tag_n does not have a node $n' \in N$ with tag tag_n as an ancestor in the tree.*

If the graph of the schema contains cycle, then every node in that cycle is possibly a recursive node. Note that if the XML document is strongly independent, it does not mean that the corresponding XML schema is strongly independent. The strong independence of the XML document is the necessary condition for the XML schema strong independence.

However, in the case that the XML document is strongly independent, but the corresponding XML schema is not, we may consider change of the XML schema. Strong independence of an XML schema can be a strong feature in the context of the efficient processing of a TPQ. An XML document valid to an XML schema without a cycle has to be strongly independent since it cannot contain any recursive nodes.

Example 4.3.2 *(Strongly independent XML schema) Figure 4.5(a) shows the example of the XML schema graph corresponding to the XML document in Figure 4.4. The graph contains cycle between the nodes section and text and due to this fact*



(a) (b)

Figure 4.5: (a) XML schema graph of the XML document in Figure 4.4 (b) Refactored XML schema

the XML schema is not strongly independent even though the corresponding XML document is strongly independent.

We can consider refactoring of the XML schema in Figure 4.5(a) into the form in Figure 4.5(b) if we know that the recursive occurrence of the section element in the XML document is limited. We can avoid the XML document recursiveness in this way and preserve the strong independence of the XML schema.

4.4 Strongly Independent Set of Tags under LPS

The strong independence for an XML document is still quite restrictive and we do not need to have an optimal algorithm for any TPQ Q over the XML document. That is a main motivation for definition of a strongly independent XML document for the SI set.

Definition 4.4.1 *(Strongly independent XML document for the SI set) Let us have the SI set of tags from the XML document D such that $SI \subseteq TGS_D$. We say that the XML document is strongly independent for the SI set if any tag $\in SI$ is non-recursive.*

The SI set is an abbreviation of a strongly independent set, where the SI set contains the set of non-recursive nodes in the XML document.

Theorem 4.4.1 *(Strongly independent XML document for the SI set) Let D be an strongly independent XML document for the SI set. Then any valid TPQ Q for the XML document D, such as every query node $q \in checkingnodes(Q)$ has the tag $tag_q \in SI$, satisfies the strong independence condition.*

Proof. (sketch) The proof is similar to that of Theorem 4.3.1. The only difference is that only the query nodes $q \in checkingnode(Q)$ have to be non-recursive. Clearly the strong independence condition have to be satisfied only for query nodes $q \in checkingnode(Q)$ according to Theorem 2.3.1 and we are done. □

The SI set can be an important parameter of the XML document. Having the SI set for an XML document we can find queries which possibly do not satisfy the strong independence condition.

In the same manner as in the previous section we can define the *strongly independent XML schema for the SI set*. Corresponding graph to a schema can contain a cycle, but none of the nodes from the SI set is in the cycle.

Example 4.4.1 *(Strongly independent XML document for the SI set)*

Figure 4.6(a) shows the example of the strongly independent XML document D for the SI set, where the SI set contains all tags from the TGS_D set without the section tag. The corresponding strongly independent XML schema S for the SI set is in Figure 4.6(b). We can see the cycle with the section tag.

We can simply reveal the TPQs which possibly do not satisfy the strong independence condition. If the section tag is not included in query nodes $q \in checkingnode(Q)$, then the LPS holistic algorithm is certainly optimal for such a TPQ Q.

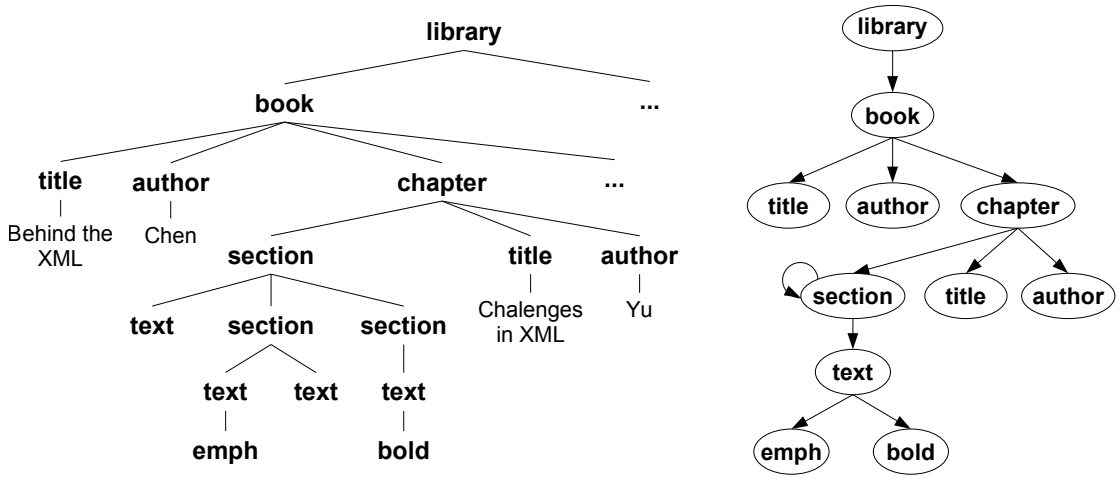


Figure 4.6: (a) XML document (b) Corresponding XML schema graph

Chapter 5

Efficient Processing of the Content-based Queries

In the previous chapters we focus mainly on queries without content. However, users often need also content-based TPQs like $//book[./author = 'Svatopluk\ Jan']/title$. We can simply store the node content with the node label in the stream when a holistic approach is considered. Then we have to sequentially scan the whole stream in order to retrieve nodes with the specific content.

Usage of a value index is another solution for content-based queries processed with a holistic algorithm. However, the value can be distributed in the whole stream and the streams without the content condition in the query have to be fully scanned anyway. For example, the TPQ Q depicted above could use the value index with an *author* stream, but the *title* stream has to be fully scanned.

We describe a structural index for a path labeling scheme, where the key of the index involves also a content of the node. We can utilize high selectivity of a content condition and use it to speed-up the TPQ processing. We use a multi-dimensional data structures for this purpose, namely the R*-tree [23] data structure and its variant the Signature R*-tree [32].

This structural index [B4] then can be used in a cost-based optimization method for content-based TPQs described in [B3].

We can expect a higher occurrence of content-based queries in the case of data-centric XML documents.

In Section 5.1 we describe the multi-dimensional approach to XML indexing. In Section 5.2 we describe similar indexing method using the B^+-tree and in Section 5.3 we propose the cost-based selection algorithm which is capable of using the indexing methods described in the previous sections.

5.1 Multi-dimensional Approach to XML Indexing

We introduce an indexing method that is significantly different from the stream model introduced in Chapter 2. Our indexing method is designed for a path labeling scheme and it allows to index node labels with a variable length. This solution involves mainly indexing method and we can utilize the TJDewey algorithm for a query processing. We utilize the multi-dimensional data structures for the purpose of XML data indexing [29, 30, B4].

Basically we can utilize any streaming scheme, but the LPS scheme has many advantages as shown in Chapter 2. The main advantages are shorter streams and larger optimality for the holistic approaches. Therefore, we use the LPS in order to achieve the best performance.

In Section 5.1.1 we introduce the indexing schema of stored data. In Section 5.1.2 we describe the TPQ processing and in Section 5.1.3 the underlying data structures are described.

5.1.1 Indexing Schema

This indexing schema is designed for a labeled path streaming scheme and a path labeling scheme.

Let us have the XML tree X with the attribute $Height$ which is the maximal length of the labeled path in the XML tree X. The labeled path lp of a node n is represented by a number $lpnumber_n$ here. We get the labeled paths' numbers as a result of the stream pruning algorithm described in Chapter 3. Normally the stream pruning returns pointers to streams, therefore, there is no significant difference.

The node n content is represented by a number $valuenum_n$. These numbers are stored in a term index, which provides mapping from a string to a number.

Node's label can be expressed in a form of the vector ($l1_n$,$l2_n$,. . . ,lk_n).

The schema of the tuple for a node $n \in X$ is as follows:

$$tuple_n = \underbrace{(lpnumber_n, valuenum_n, l1_n, l2_n, \ldots, lk_n, 0, \ldots, 0)}_{Height+2}$$

Example 5.1.1 *(Indexing Schema) Let us consider the XML tree X in Figure 5.1(a) and its corresponding DataGuide in Figure 5.1(b). The XML tree is labeled using the Dewey order labeling scheme. The node content is represented by italic text in Figure 5.1 and the corresponding* $valuenum_n$ *is above every node content. For*

example, the node c_2 has the content 'M.15' and $valuenumber_{c_2} = 2$. The labeled path numbers are denoted in the DataGuide. It can be seen that $Height = 5$.

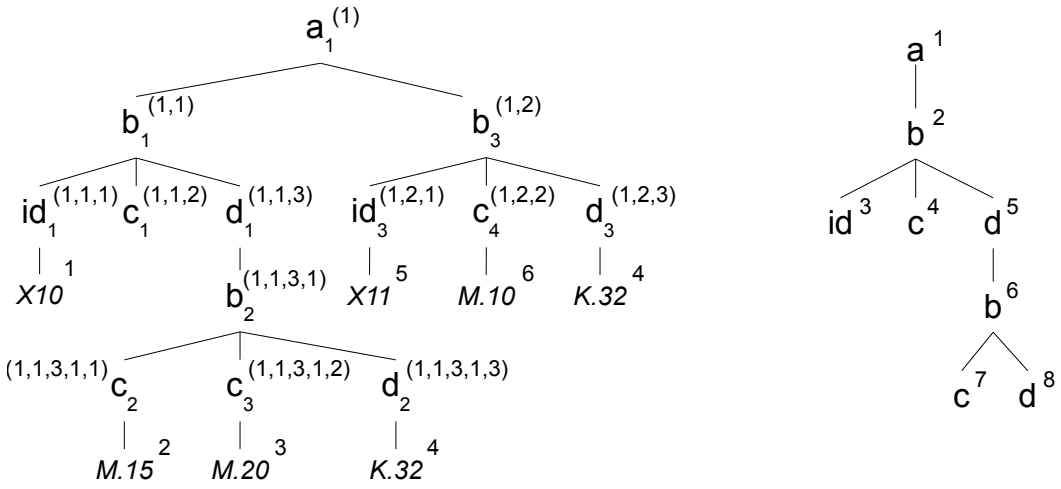


Figure 5.1: (a) the XML tree (b) the corresponding DataGuide

Following our indexing schema the tuple for the node c_4 is $tuple_{c_4} = (4, 6, 1, 2, 2, 0, 0)$. The tuple for the node b_3 is $tuple_{b_3} = (6, 0, 1, 1, 3, 1)$.

Obviously the label of the node n can be easily extracted from the tuple $tuple_n$. Therefore, we sometimes use the term *node label* or *node* instead of *tuple*.

We generate tuple from every node in the XML tree and store these tuples in a data structure capable of indexing every attribute of the tuple. The B$^+$-trees would be a very expensive solution, and therefore we use a multi-dimensional data structure (see Section 5.1.3). The multi-dimensional index can serve as an index and also as a storage of the tuples, because we store the whole tuples in the data structure.

5.1.2 Query Processing

The multi-dimensional data structure allows us to query any attribute of the tuple by a range query. As a result we obtain a set of tuples within the range specified by a range query.

The indexing schema described in the previous section enables two different techniques of a query processing. Both techniques utilize labeled paths, and therefore

the stream pruning has to be the first strep of a TPQ processing. We obtain the set of labeled paths numbers (not streams as it was in the case of holistic approaches) and the *soln*() function can be utilized as well. Due to the fact that we work with a path labeling scheme, we retrieve only nodes corresponding to the leaf query nodes (similarly to the TJFast or TJDewey algorithms).

Merge Join

The first technique is similar to a holistic approach of a query processing. We retrieve tuples corresponding to one labeled path by one range query and we treat these results as a stream. In that case the TJDewey algorithm can be applied. This idea is described more thoroughly in Section 5.2.3.

The range query for a labeled path with $lpnumber_n$ is as follows:

$$(lpnumber_n, 0, 0, \ldots, 0) : (lpnumber_n, max, max, \ldots, max)$$

If we search nodes with some particular value $valuenum_n$, we can specify this value in the range query as follows:

$$(lpnum_n, valuenum_n, 0, \ldots, 0) : (lpnum_n, valuenum_n, max, \ldots, max)$$

It can be observed that the second range query significantly reduce the searched space and all retrieved nodes have the node content represented by $valuenum_n$.

Example 5.1.2 *(Merge Join) Let us consider the XML tree X in Figure 5.1(a) and the TPQ* `/a/b[./c = 'M.10']//d`*. After the stream pruning we obtain the following labeled path numbers:* $\{1\}$ *for the query node a,* $\{2\}$ *for the query node b,* $\{4\}$ *for the query node c, and* $\{5, 8\}$ *for the query node d. These labeled path numbers are connected through soln() multiset. As a next step we retrieve the content number* 6 *from the term index (value 'M.10').*

We have all informations necessary for a range query searching. We run three range queries for every labeled path corresponding to the leaf query nodes.

Query node c:

$$(4, 6, 0, 0, 0, 0, 0) : (4, 6, max, max, max, max, max)$$

Query node d:

$$(5, 0, 0, 0, 0, 0, 0) : (5, max, max, max, max, max, max)$$

$$(8,0,0,0,0,0,0):(8,max,max,max,max,max,max)$$

We obtain only one node label $(1,2,2)$ *for the query node c and node labels* $(1,1,3),(1,1,3,1,3),(1,2,3)$ *for the query node d. These results can be merged utilizing the TJDewey algorithm, because we use the pruned labeled paths (analogously to pruned streams). Therefore, we can extract MB set for every labeled path as described in Section 2.4.2 and use it during the TJDewey algorithm processing.*

We get the node label $(1,2,3)$ *as a result of the TJDewey join and this is the query result.*

Progressive Join

The second technique is based on context nodes. If we have a label of the node n and we want to retrieve its ancestors, we can utilize range query again.

Let us have a node n with the label ($l1_n$,$l2_n$,...,lk_n). We search its descendant nodes with the lp_d labeled path number and we retrieve these nodes from the multidimensional data structure using the following range query:

$$(lp_d,0,l1_n,\ldots,lk_n,0,\ldots,0):(lp_d,max,\underbrace{l1_n,\ldots,lk_n,max,\ldots,max}_{length\ of\ the\ searched\ labeled\ path},0,\ldots)$$

We can further specify the second dimension of the range query, if we search nodes with a specific content.

This type of query processing can be particularly useful when we have small number of context nodes. It is not necessary to scan all nodes corresponding to the labeled path, but we search only nodes with respect to these context nodes.

Example 5.1.3 *(Context nodes) Let us consider the XML tree X in Figure 5.1 and the TPQ* `/a/b[./c = 'M.10']//d` *again. We have the same set of labeled path numbers and the content number* 6 *is also the same.*

We start with the range query for a query node c:

$$(4,6,0,0,0,0,0):(4,6,max,max,max,max,max)$$

We obtain the node label $(1,2,2)$*. Using the* $MB((1,2,2),b)=\{(1,2)\}$ *we reveal that the node* $(1,2,2)$ *has only one parent node* $(1,2)$ *corresponding to query node b. That is the context node for the query node b and we want to find all descendant*

nodes lying on labeled paths with the numbers 5 *and* 8*. We retrieve these nodes using following range queries utilizing the context node* $(1, 2)$ *as follows:*

$$(5, 0, 1, 2, 0, 0, 0) : (5, max, 1, 2, max, 0, 0)$$

$$(8, 0, 1, 2, 0, 0, 0) : (8, max, 1, 2, max, max, max)$$

These range queries retrieve only node the label $(1, 2, 3)$ *and that is the result of the TPQ processing.*

These range queries are more specific, and therefore they search significantly smaller size of the tree. On the other hand, they have to be processed for every context node, due to this fact they are efficient only for a limited number of context nodes.

In Section 5.3 we propose a simple cost-based selection algorithm capable of using both types of a joins. This selection algorithm can be applied also in the case of the multi-dimensional approach.

5.1.3 Index Data Structures

Due to the fact that we model XML document as a set of points in a multi-dimensional space, we apply multi-dimensional data structures like the (B)UB-tree [3] and the R-tree [23].

An important issue concerning the multi-dimensional index is a variable length of vectors. Documents with the $Height = 10$ may exist, but documents with $Height = 36$ as well (see [69]). A naive approach is to align the dimension of space to the maximal length of the path in the XML document. For example, points of dimension 5 will be aligned to dimension 36 using the zero in the 6^{th}–36^{th} coordinates. This technique increases the size of an index and the query processing overhead as well. In [31] BUB-forest data structure was introduced. This data structure allows us to store tuples with various dimensions efficiently. BUB-forest contains several BUB-trees, each of them store vectors of different dimensions. We can apply the same approach for other data structures, e.g., R-trees. This approach yields a very efficient RPE processing which is not dependent on the XML tree depth.

The range queries processed in the multi-dimensional approach are called *narrow range queries*. Points defining a query box share some of the same coordinate values, whereas the size of the interval defined by other coordinates is near to the size of the space's domain. Note: the regions intersecting a query box during the processing of a range query are called *intersect regions* and regions containing at least one point

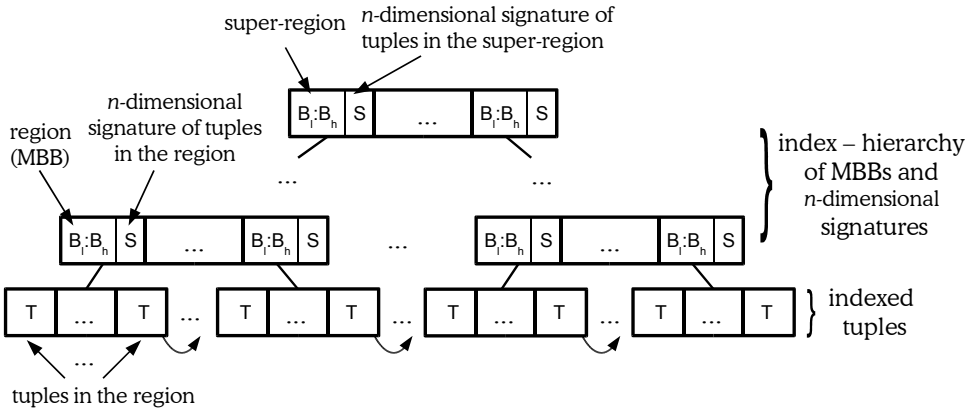


Figure 5.2: A structure of the Signature R-Tree.

of the query box are called *relevant regions*. Many irrelevant regions are searched during the processing of the narrow range query in multi-dimensional data structures. Consequently, a ratio of relevant and intersect regions, so called *relevance ratio*, is significantly decreasing with an increasing dimension of an indexed space.

In [32] Signature R-tree data structure was introduced. This data structure enables efficient processing of the narrow range queries. Items of inner nodes contain the definition of a (super)region and an n-dimensional signature of tuples is included in the (super)region (see Figure 5.2). A superposition of tuples' coordinates by the operation `OR` creates the signature. The Operation `AND` is used for better filtration of irrelevant regions during processing of the narrow range query. Other multi-dimensional data structures (e.g., (B)UB-tree) are possible to extend in the same way. In [30] we show that when we process a narrow range query, the Signature R-tree is more efficient that the Signature (B)UB-tree.

5.2 Relational Approach

In the previous section we describe the multi-dimensional approach to XML indexing and two different views on a twig query processing. In work [11] similar solution for a content-based query processing based on a B^+-tree was introduced. The authors of [11] compare two indices which use the LPS and a path labeling scheme. The first index, the *ROOTPATHS* index, is able to process twig queries using the merge join.

The second index, the *DATAPATHS* index, is able to process a query path by utilizing previous query results (utilizing context nodes in our language). Consequently, the DATAPATHS index applies a progressive join algorithm. These indices can outperform each other depending on a query. However, there is no general proposal that can help to decide which index should be used to achieve the best query processing performance.

In Section 5.2.1 the indexing schema using the B^+-trees is proposed. In Section 5.2.2 we describe the query processing using the merge join and the progressive join when the B^+-tree indices are considered.

5.2.1 Indexing Schema

The labeled path is represented by a number in the same way as in the multidimensional approach. We have to utilize the stream pruning to obtain the labeled paths' numbers. We also use the term index mentioned in Section 5.1.1 for a mapping of a node content into the number.

Inverted List

Let us consider the relation

$$(\underline{\text{Labeled path}}, \underline{\text{Node value}}, id)$$

,where *id* is a label of a node corresponding to the *labeled path* and with the *node value*. We store every node label in this relation and create inverted list, where the key is built by a concatenation of the labeled path and the node value.

Example 5.2.1 *(Inverted List) The example of the inverted list for the XML document in Figure 5.3 is shown in Table 5.1. We left the whole labeled paths and the node contents in the table for the easier orientation.*

The node value is a part of the key used for a faster execution of content-based queries. On the other hand, in the case of the inverted list, where the node content is a part of the key, the input lists from the inverted list need to be additionally sorted when we process query paths without the content.

Example 5.2.2 *(Node Sorting) Let us have the document in Figure 5.3 and the TPQ* `//b/c='u'`*. We can directly access the node (1,1,1) without scanning the whole list of nodes for the* `/a/b/c` *labeled path.*

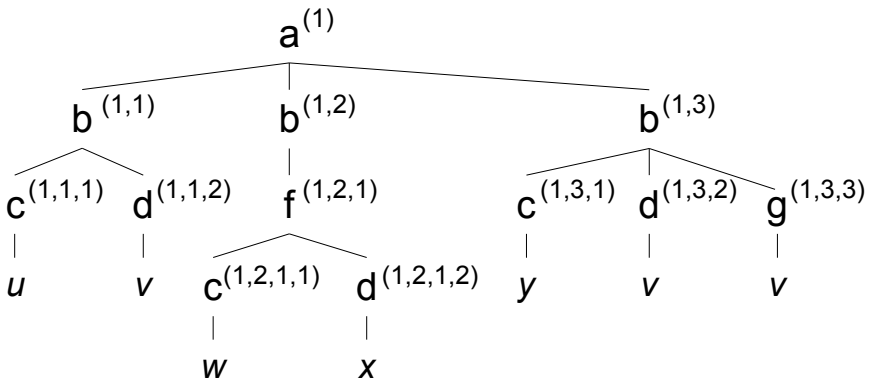


Figure 5.3: Example of an XML tree.

Table 5.1: Example of the inverted list for the XML document in Figure 5.3

Key	**List of node *id*s**
(a/b,null)	(1,1), (1,2), (1,3)
(a/b/c,u)	(1,1,1)
(a/b/c,y)	(1,3,1)
(a/b/d,v)	(1,1,2), (1,3,2)
(a/b/f,null)	(1,2,1)
(a/b/f/c,w)	(1,2,1,1)
(a/b/f/d,x)	(1,2,1,2)
(a/b/g,v)	(1,3,3)

If we consider the TPQ `//b/c` *then the sets* $\{(1,1,1)\}$ *and* $\{(1,3,1)\}$ *corresponding to the labeled path* $a/b/c$ *has to be merged together in order to obtain the sorted result for this TPQ. It is simple in this example, however, the sets can be considerably larger.*

This issue shows that this approach is mainly suitable for content-based queries. Of course, we can omit the node content from the key for some labeled paths. That would be particularly useful if we do not expect queries with a content condition bounded with these labeled paths. However, this is out of scope of this thesis.

Node Index

Let us consider the relation

$$(\underline{\text{Context node label}}, \underline{\text{Labeled path}}, \underline{\text{Node value}}, Did)$$

, where *Did* is a label of a descendant node to a context node corresponding to a *labeled path* and with the *node value*. We store all descendant nodes to every node in an XML document. The node index is a B^+-tree index, where the key is made by a concatenation of the first three attributes of the relation.

Example 5.2.3 *(Node Index) Let us consider the XML tree in Figure 5.3 we provide the example of a corresponding node index in Table 5.2.*

Key	**List of node *id*s**
((1,1),a/b/c,u)	(1,1,1)
((1,1),a/b/d,v)	(1,1,2)
((1,2),a/b/f,null)	(1,2,1)
((1,2),a/b/f/c,w)	(1,2,1,1)
((1,2),a/b/f/d,x)	(1,2,1,2)
((1,2).1,a/b/f/c,w)	(1,2,1,1)
((1,2).1,a/b/f/d,x)	(1,2,1,2)
((1,3),a/b/c,y)	(1,3,1)
((1,3),a/b/d,v)	(1,3,2)
((1,3),a/b/g,v)	(1,3,3)

Table 5.2: Example of the node index for the XML document in Figure 5.3

Therefore, we index the whole subtree of every node. Similar index is called the DATAPATH in work [11]. There is one significant difference between the DATAPATH and our node index. We do not include subtree for a root node of XML tree in a node index, because this subtree is already stored in an inverted list. In this way we 'split' the DATAPATH into the inverted list and the node index.

The maximum number of records in a node index is $D \cdot N$, where D is the average depth of an XML tree and N is the number of nodes. The multi-dimensional index described in Section 5.1 store only one record per node, therefore, the number of records in that index is N.

5.2.2 Query Processing

We consider two types of join algorithms: merge and progressive join operations similarly as they was described in Section 5.1. In Section 5.1 we mainly focus on a description of node labels retrieval in the context of the multi-dimensional data structure. We only mention that we should process the merge join using the TJDewey algorithm, but this statement should be described more thoroughly. We describe the issue of merge join in Section 5.2.3.

The stream pruning is the first step of a TPQ processing. We use the stream pruning in the same way as it is described in Section 5.1.1.

5.2.3 Merge Join

The *merge join* is based on a merge algorithm that joins labels retrieved from the inverted list (or from the multi-dimensional data structure). We always merge node labels corresponding to two query paths and we produce an intermediate result corresponding to joined query patterns. This kind of query processing is very similar to the binary structural join described in [34, 71, 1].

When we are using the binary joins we have to always select appropriate query plan. We utilize concept of query plans introduced in [68], but there is one significant difference. The binary structural joins process the query nodes, but the merge join process the query paths. These difference is due to the labeling scheme used. The binary structural joins are designed for an element labeling scheme, whereas, the merge join is proposed for a path labeling scheme. That is also a reason why we use a TJDewey algorithm for the merge join, because the other binary joins approaches always consider only the element labeling scheme.

Example 5.2.4 *(Merge Join versus Structural Join) Let us have the TPQ* `//a/b[./c and .//d]/f`*. According to the article in [68] there is number of possible query plans using binary structural joins. We can see example of one query plan using the binary structural joins in Figure 5.4(a). We produce intermediate result after each structural join, which is used in an additional query processing. The details about the structural join algorithms can be found in [1].*

We can use the same concept of the query plans in the case of the merge join. We can observe one possible query plan using the merge join in Figure 5.4(b). We can simply use the TJDewey algorithm as a merge join algorithm. Therefore, we are not using this algorithm as a holistic algorithm, but we merge results only for the two leaf query nodes (two query paths).

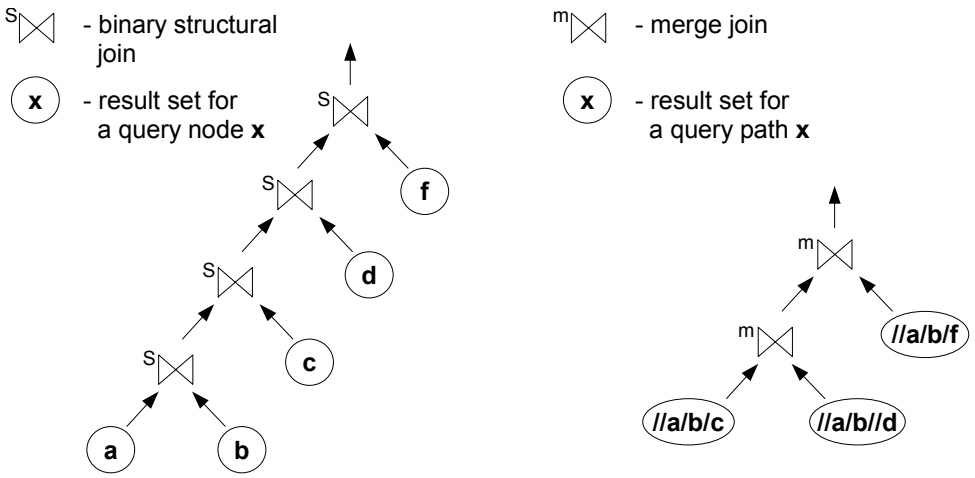


Figure 5.4: (a) Query plan using structural joins (b) Query plan using the merge joins

Selection of an appropriate query plan is a main issue of the Section 5.3.

It can be observed that the terms 'query path' and the 'leaf query node' have the same meaning in the path labeling scheme approach. Due to the fact that we work only with streams in leaf query nodes we have to select from these streams only node labels corresponding to the whole query path.

Example 5.2.5 *(Merge Join) Let us have the TPQ* `//b[.//c and ./g]/d` *and the document in Figure 5.3. We process the stream pruning and we get the set of labeled paths corresponding to every query node. The labeled paths for the leaf query nodes are as follows:* $\{a/b/c\}$ *for the query node c,* $\{a/b/g\}$ *for the query node g, and* $\{a/b/d\}$ *for the query node d.*

We can, for example, first process the merge join of the query paths `//b//c` *and* `//b/g`*. We use the TJDewey for that purpose and we join set (stream in the TJDewey terminology)* $\{(1,1,1),(1,3,1)\}$ *corresponding to the query node c and set* $\{(1,3,3)\}$ *corresponding to query node g. Obviously, we are working only with the leaf query nodes. As a result of the merge join we obtain the label* $(1,3)$ *and we merge join this intermediate result with the query path* `//b/d`*. The query path* `//b/d` *is an output query path, which means that we should process this query path at the end of the query processing. Finally we get the result node* $(1,3,2)$ *after the merge join of the intermediate result set* $\{(1,3)\}$ *with the set* $\{(1,1,2),(1,3,2)\}$*.*

Progressive Join

We describe how the progressive join is processed in the case of an inverted list and a node index. We can use context nodes from the previous intermediate result and process a query over node index for the every context node. This solution was already described in the context of the multi-dimensional approach in Section 5.1.2.

We have the same input information: labels of the context nodes, labeled paths of searched nodes, and optionally we can also have a content number of searched nodes. We get the same node labels as if the merge join is processed.

Example 5.2.6 *(Progressive join)*

Let us consider the TPQ `//b[.//c and ./g]/d` *from Example 5.2.5 again. We have the same sets of labeled paths after stream pruning.*

We process this query using the progressive join. First, we retrieve the node labels corresponding to the query node g from the inverted list. We get the node label $(1, 3, 3)$ *and we can extract the only one context node label* $(1, 3)$*. Then, we continue with progressive join between the query paths* `//b/g` *and* `//b//c`*. We have the context node* $(1, 3)$ *and the labeled path* $a/b/c$ *of class c. In Table 5.2 we can see that the index contain the result node label* $(1, 3, 1)$ *(eighth row). Therefore we get the same context node* $(1, 3)$ *as a intermediate result of the progressive join of the query paths* `//b//c` *and* `//b/g`*.*

We continue with another progressive join with the same context node and labeled path `a/b/d`*. We get the node label* $(1, 3, 2)$ *which is also the result node.*

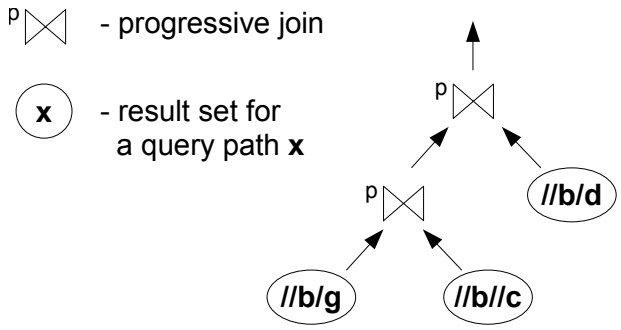


Figure 5.5: Query plan using the progressive joins

We can see the corresponding query plan in Figure 5.5.

5.3 Cost-based Join Selection

In this section we introduce the cost-based optimization technique for a join selection during a TPQ processing. We show that the knowledge of the result size can help us to choose a good query evaluation strategy. We show this issue on indices using B^+-tree, because their complexity can be easily estimated. We can use the cost-based optimization for the multi-dimensional approach as well since it provides both types of the join algorithms.

5.3.1 Join Selection Algorithm

In Algorithm 11, we propose the cost-based join selection algorithm [B3]. An important operation of this algorithm is the building of an appropriate query plan in Line 1. There are some works that support this operation [67, 46]. Authors often provide algorithms and data structures for an estimation of query-path result size. Query paths are sorted according to the result size or the size of a join result. In other words, we often process the query-path with the highest selectivity first. This is the starting point of our join selection algorithm.

`SelectJoin` is the core function of this algorithm (see Line 4). This function selects the most appropriate join operation. In other words, this selects which query path is processed by `MergeJoin` (see Lines 5–6) and `ProgressiveJoin` (see Lines 9–10), respectively. In the next Section we introduce a theoretical model for an implementation of the `SelectJoin` function.

5.3.2 A Cost-based Model for the Selection

We use the relational indices described in Section 5.2 for our cost model. We use the $\mathcal{QR}_x$ symbol in the following text representing the result set of the x-th query path.

Due to the fact that a path join is processed for two query-path results, we offer the cost of the $\mathcal{QR}_1 \otimes \mathcal{QR}_2$ operation. Let m be $|\mathcal{QR}_1|$ and n be $|\mathcal{QR}_2|$, due to the previously published query result estimations we suppose $m \leq n$.

We must distinguish the processing cost of query path with the content, SP^C, and a query path without a content, SP^S. The SP^C is processed by a search in the inverted list. Let the N_{lp} be the number of different node contents belonging to the labeled path lp. Since the N_{lp} is usually greater than one in the case of SP^S we have to sort elements retrieved from the inverted list. Consequently, costs of simple-path query processing, `ProcessQueryPath` function in Algorithm 11, follow:

Algorithm 11: A Cost-based Join Selection Algorithm

```
   input  : A ℚ query
   output: A set of nodes ℛ^ℚ
 1 int[] rsa ← BuildQueryPlan (ℚ.P);
 2 Node[] result1 ← ProcessQueryPath (Q_{P_rsa[0]});
 3 for i ← 1 to |ℚ.P| do
 4     if SelectJoin (ℚ_{Pi−1}, ℚ_{Pi}) then
 5         Node[] result2 ← ProcessQueryPath (Q_{P_rsa[i]});
 6         result1 ← MergeJoin (result1,result2);
 7     else
 8         foreach n ∈ result1 do
 9             result1 ← ProgressiveJoin (result1, n, Q_{P_rsa[i]});
10         end
11     end
12 end
13 return result1
```

$$\begin{array}{lll} SP_1^C.Cost = & \log\sigma + m & \text{Inverted list search} \\ SP_1^S.Cost = & \log\sigma + m + & \text{Inverted list search} \\ & m \cdot \log m & \text{Sorting of the result} \end{array}$$

where σ is the number of keys in the inverted list: the sum of the N_{lp} for all labeled paths in the XML document.

In the case of merge join (the `MergeJoin` function in Algorithm 11) we use the TJDewey algorithm with $O(m+n)$ complexity. The cost is the following:

$$M.Cost = SP_1.Cost + SP_2.Cost + m + n + |\mathcal{R}|$$

The `ProcessQueryPath` function is processed twice due to the fact that $\mathcal{QR}_1$ and $\mathcal{QR}_2$ are retrieved from an inverted list. $m+n+|\mathcal{R}|$ express the TJDewey algorithm complexity. Of course, there is an optimality issue of the TJDewey algorithm as well, however, we can approximate the complexity here. The $|\mathcal{R}|$ expresses the result size.

In the case of the progressive join (the `ProgessiveJoin` function in Algorithm 11) we propose the following cost:

$$P.Cost = SP_1.Cost + m \cdot \log\gamma + |\mathcal{R}|,$$

where γ is the number of keys in the node index. This number is approximately $N \ldots L$, where N is a number of nodes in the XML tree and L is average depth of the XML tree. Obviously, the `ProcessQueryPath` function is processed in the first

phase. In the second phase, the query is processed in the node index for the every context node retrieved in the first phase.

So continuing with our description:

$$\begin{aligned} M.Cost &= \log\sigma + m + \theta(SP_1) + |\mathcal{R}| + \\ &\quad \log\sigma + n + \theta(SP_2) + m + n + |\mathcal{R}|, \end{aligned}$$

$$\begin{aligned} P.Cost &= \log\sigma + m + \theta(SP_1) + |\mathcal{R}| + \\ &\quad m \cdot \log\gamma, \end{aligned}$$

where $\theta(SP)$ is the cost of a sorting function and it its defined as follows:

$$\theta(SP) = \begin{cases} m \cdot \log m & SP = SP_1 \text{ and it is not the content-based query} \\ n \cdot \log n & SP = SP_2 \text{ and it is not the content-based query} \\ 0 & \text{the query is the content-based query} \end{cases}$$

One `ProcessQueryPath` function is processed for the SP_1 evaluation in the case of both join operations. When we want to develop a selection function, we have to compare the following costs:

$$\begin{aligned} M.Cost' &= \log\sigma + n + \theta(SP_2) + m + n \\ &= \underline{2 \cdot n + m + \theta(SP_2) + \log\sigma}, \end{aligned}$$

$$P.Cost' = \underline{m \cdot \log\gamma}.$$

Due to the fact that only the merge join depends on the n value, these costs result in the following issue.

Issue

The progressive join is more efficient than the merge join when

1. $m << n$.

2. m is rather low.

We learn some parameters based on statistics provided in real XML documents. For example, the XMARK collection with factor 10 includes: $|\mathbb{X}.LP| = 1,234$, $\sigma = 3,105,344$, $|\mathbb{X}.P| = 12,456,234$, and $\gamma = 63,495,257$. Consequently, $\log \sigma = 7$ and $\log \gamma = 8$. We can not assess the overall cost precisely, due to the fact that we use atomic operations with incomparable costs:

- The cost of one step in a sorting function may be lower than the one step in a persistent tree search.
- The cost of searching in the inverted list may be lower than the search in the node index, due to the fact that random accesses are processed in the case of the node index.

However, our experiments prove that **Issue** is robust enough for the operation costs. We recommend to use a progressive path-join if $m < n/150$. Obviously, we must know m, n, and $|\mathcal{R}|$ values. In other words, we must know query-path results and the join result. There are works that enable estimation of these values [67, 46, 68, 47].

5.4 Experimental Results

In our experiments[1], we test our join selection algorithm on the XMARK collection with the factor 10 [53]. The collection contains one file, 1.1 GB in size, including 33 million nodes.

Characteristics of these indices are shown in Table 5.3. We observe that the node index is six times bigger than the inverted list. It is close to the average depth of the XMARK XML document, see Section 5.2.1. Dewey order enlarges these indices, however, it is possible to resolve this issue as shown in work [24], with a simple prefix compression and variable-length code.

Our own implementation of persistent data structures[2] was used in our experiments. We also apply other indices such as term, labeled path and result estimation indices. The term index maps the terms into integer values and the labeled path index maps the labeled paths into integer values. Since the processing time over these indices was rather short for a query, we do not present these results. Moreover, efficiency of these indices does not influence the join algorithm comparison due to the fact that the same operations over these indices are processed for a query.

[1] The experiments were executed on an Intel Pentium 4 2.4 Ghz, 1GB DDR400, with Windows XP.

[2] ATOM – `http://arg.vsb.cz/`

Table 5.3: Characteristics of indices

	Inverted list	**Node Index**	**LP Index**	**Term Index**
Index size	376 MB	2.45 GB	66 KB	268 MB
Page size	2 kB	2kB	in-memory	4kB

As usually, we set a data structure cache to minimum for processing all tested query (around 1 MB for each index). Therefore, query processing time was not affected by a 'hot cache'. We mainly focused on disk access cost (DAC) reduction.

Table 5.4: Evaluated XPath queries

	XPath query	**Query Path Result**	**Result Selectivity**
$\mathbb{Q}_1$	`//person[./profile/age]` `/@id`	64,471 255,000	64,471
$\mathbb{Q}_2$	`//namerica/item[./location='Canada']` `/name`	146 100,000	146

$\mathbb{Q}_3$	`//person[.//country='United States'`
	`and ./profile/business='Yes']`
	`/@id`
		96,148
		63,802
		255,000
			63,802

$\mathbb{Q}_4$	`//namerica/item[./location='Germany'`
	`and ./payment='Creditcard']`
	`/name`
		74,917
		6,085
		100,000
			8

$\mathbb{Q}_5$	`//asia/item[./location='United States'`
	`and ./@id='item5501']`
	`/name`
		14,990
		1
		20,000
			1

$\mathbb{Q}_6$	`//closed_auction[./annotation/happiness='9'`
	`and ./quantity='1']`
	`/price`
		9,478
		89,230
		97,500
			8,697

$\mathbb{Q}_7$	`//open_auction[./bidder/increase='12.00'`
	`and ./privacy='No']`
	`/type`
		28,106
		30,316
		120,000
			5,947

In our test, we process a query many times in all join type permutations. To decrease the total number of possible query executions, we always join query paths with the highest selectivity first. In this way, we find the best query plan, which is compared with processing based on only merge and progressive path-joins. These results are compared with the result of the proposed join selection algorithm.

Tested queries (see Table 5.4 and Table 5.5) include 2, 3, and 4 query paths to show how join algorithm selection affects query processing. Most of the queries con-

Table 5.5: Evaluated XPath queries

	XPath query	**Query Path Result**	**Result Selectivity**

$\mathbb{Q}_8$	`//asia/item[./quantity='1'`
	`and ./location='Thailand']`
	`/*/mail/from`
		18,455
		20
		19,296
			19

$\mathbb{Q}_9$	`/namerica/item[./quantity='1'`
	`and ./location='United States'`
	`and ./incategory/@category='category1000']`
	`/name`
		91,956
		74,917
		41
		100,000
			24

$\mathbb{Q}_{10}$	`//open_auction[./type='Regular'`
	`and ./quantity='1']`
	`/annotation[./happiness='8']`
	`//keyword`
		57,913
		110,552
		12,700
		166,870
			6,848

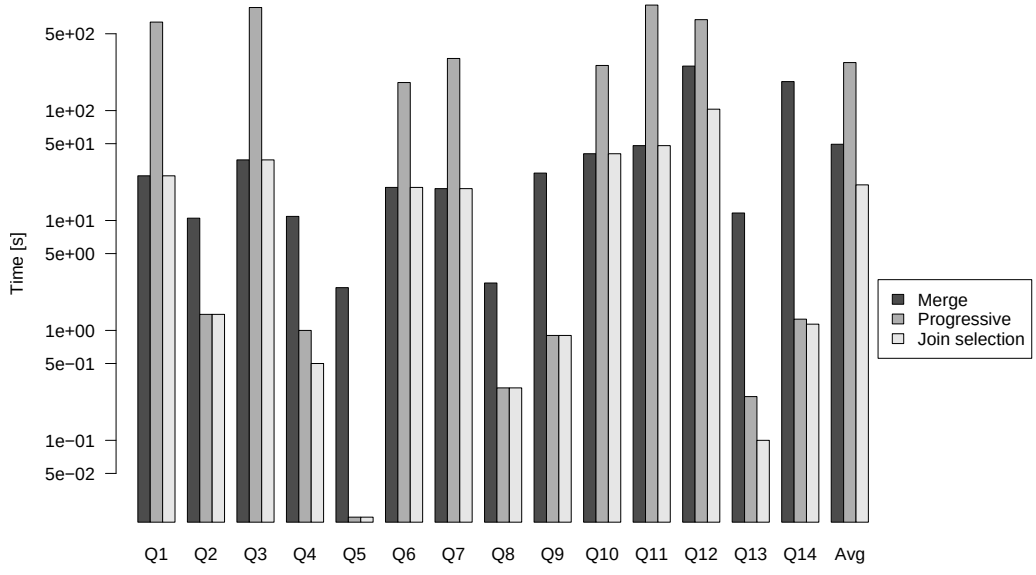
$\mathbb{Q}_{11}$	`//people/person[./profile[./age`
	`and ./gender]`
	`and ./creditcard]`
	`/@id`
		64,471
		63,954
		127,695
		255,000
			15,963

$\mathbb{Q}_{12}$	`//person[./profile[./gender='female'`
	`and ./business='Yes']`
	`and ./creditcard]`
	`/address/*`
		31,790
		63,802
		127,695
		572,596
			18,078

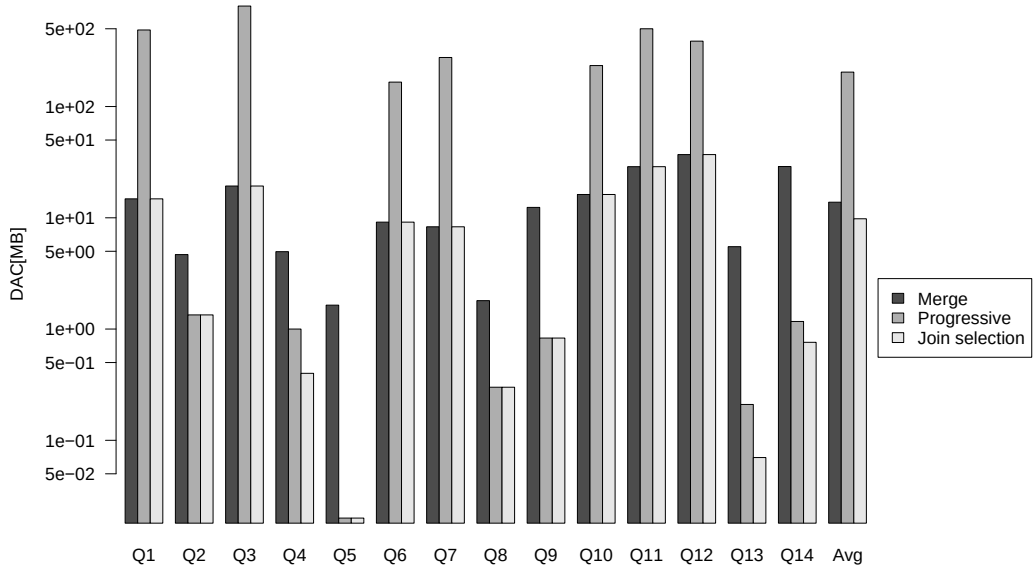
$\mathbb{Q}_{13}$	`//asia/item[./quantity='1'`
	`and ./location='Denmark' and`
	`./payment='Cash']`
	`/*`
		18,455
		27
		1,341
		215,829
			37

$\mathbb{Q}_{14}$	`//person[./profile[./gender='male'`
	`and ./education='Graduate School']]`
	`/address[./country='Germany']`
	`//*`
		32,164
		16,015
		132
		572,596
			10

tain query paths with content. We choose queries containing the query paths with low selectivity such as queries $\mathbb{Q}_1, \mathbb{Q}_3, \mathbb{Q}_7, \mathbb{Q}_{11}$, and $\mathbb{Q}_{12}$ and also queries containing at least one query path with higher selectivity such as $\mathbb{Q}_5, \mathbb{Q}_8$, $\mathbb{Q}_9$, $\mathbb{Q}_{13}, \mathbb{Q}_{14}$. Results are shown in Figure 5.6. There are queries where the merge join exceeds the progressive join, however, there are queries with contrary results. This figure shows that the join selection reaches the best result. It utilizes the power of join algorithms: merge or progressive joins are applied according to expected query-path results. The join selection exceeds approximately 20× the progressive join and the efficiency improvement is up by 50% when compared to the merge join. We observe that results prove our issues.



(a)



(b)

Figure 5.6: The evaluation of queries (a) Time (b) DAC

Query	**Query plan operations**				
Q_9	$\texttt{QP}_2$	$\texttt{P}_1$	$\texttt{P}_0$	$\texttt{P}_3$	
	41	24	24	24	
Q_{11}	$\texttt{QP}_1$	$\texttt{QP}_0$	M	$\texttt{QP}_2$	M
	63,954	64,471	32,249	127,695	15,963
	$\texttt{QP}_3$	M			
	255,000	15,963			
Q_{14}	$\texttt{QP}_2$	$\texttt{QP}_1$	M	$\texttt{P}_0$	$\texttt{P}_3$
	132	16,015	2	2	10

Table 5.6: Detailed description of the query processing using the join selection for three types of query processing situations.

In Table 5.6 we see a detailed description of the best query processing algorithm for some queries. These queries represent three query processing situations. These situations show various orders of join algorithms. The operation QP denotes a retrieval of an input list from the inverted list for i-th query path: `ProcessQueryPath` function in Algorithm 1. In order to simplify the query process we separate the input list retrieval and join algorithm. In this experiment, we propose intermediate results to show how their sizes affect the join selection. The operation M denotes the merge join among previous intermediate results and input lists. The operation $\texttt{P}_i$ denotes the progressive join using the previous result and processes the i-th query path. The table also contains the intermediate result size for each operation.

We see that results support **Issue** and the cost-based join selection model in Section 5.3. In cases of large intermediate results, as well as large query results, the merge join is optimal and the best solution applies only the merge join. This is true for queries with low query-path selectivity. On the other hand, when the intermediate results are small, the progressive join becomes more efficient than the merge join. This is true for queries containing at least one query-path with high selectivity.

There are queries such as $\mathbb{Q}_4, \mathbb{Q}_{13}, \mathbb{Q}_{14}$, where the join selection algorithm applies merge and progressive joins alternately. Moreover, we observe that the optimal query plan never performs the merge join after progressive join. When we first process query paths with the highest selectivity and intermediate results are rather small, the progressive join becomes optimal.

Chapter 6

Conclusion and Future Work

In this thesis we make a work toward an improvement of methods to a twig pattern query searching. Twig pattern query searching is considered as a core operation of any query language for XML data [54, 71, 5, 1, 21]. We analyze the state-of-the-art approaches and propose their additional improvement.

6.1 Contribution

Holistic approaches [5, 9, 36] represent important progress in a twig query pattern processing. Compared to other approaches they provide the best worst-case space, I/O, and time complexity in a case of optimal algorithm. Optimal holistic approaches do not produce large intermediate results and their performance can be furthermore supported by some structural or value index. However, the optimality of a holistic algorithm is dependent on a specific query class.

In this work we show that the optimality of a holistic approach can be guaranteed also by a structure of an XML document. We prove the optimality of a holistic algorithm with respect to characteristics of an XML document and TPQ and not only with respect to a query class. Holistic algorithm can use different streaming schemes [9]. We follow this general design of holistic approaches and describe the optimality condition which has to be satisfied by an XML document and TPQ for different streaming schemes. We call this condition a strong independence condition.

Based on this observation, we can, for example, design an XML document, where a holistic algorithm is optimal for any twig pattern query.

We observe that the strong independence condition is the most general in the case of LPS and due to this fact we design two novel holistic algorithms using

LPS. Our first new holistic algorithm, the TwigStackSorting, is an alternative to the iTwigStack+LP algorithm. It does not have to search the lowest stream in the query node sequentially and, due to this feature, it outperforms the iTwigStack+LP in the most of considered queries. The second new algorithm, the TJDewey, can utilize any path labeling scheme (e.g., Dewey Order), and therefore it can work only with leaf query nodes. The Dewey Order enables the easier processing of an update comparing to the containment labeling scheme and to the Extended Dewey labeling scheme. The TJDewey overcomes other approaches in terms of the main memory run and number of disk accesses. The TJDewey does not have to extract the labeled path from every label if compared to the TJFast.

There is a drawback of an LPS approach in a form of the stream pruning phase. We have to find relevant labeled paths (streams) as a first step of a query processing. We basically search for labeled paths in a DataGuide tree and this is called stream pruning a in the case of holistic approaches [9]. DataGuides can be large and complex trees in the case of real XML documents. Consequently, a trivial DataGuide search is time and memory consuming. We introduce algorithms for the efficient DataGuide TPQ search which result in the pruned labeled paths. We adopt holistic approaches for this search. We introduce an algorithm capable of enumeration of the *soln* set from the hierarchical stacks of the Twig^{2}Stack algorithm. Furthermore, we introduce a fusion of the TwigStack and the Twig^{2}Stack using queues and utilize advantages of both approaches. Our experiments show that the stream pruning based on holistic approaches is more robust than the original stream pruning algorithm proposed [9]. The result stream pruning approach can be used in any algorithm using labeled paths to speed-up the query processing.

The holistic algorithm can be viewed as an operator over query nodes producing the final result. Experimental results show that an application of one join operation for the entire twig query processing is not always the most efficient solution. We introduce two structural indices capable of a query processing in a navigational fashion. The first, multi-dimensional approach, stores node labels in a multi-dimensional data structure like R-tree. The second index uses the traditional B$^+$-tree index. We describe two types of join operation in a context of these indices. The merge join is a binary structural join for a path labeling scheme and the progressive join is a navigational processing based on context nodes. We propose theoretical model for a cost-based selection of an appropriate join operation. We can achieve significant improvements during the query processing comparing to an approach using only the merge join.

We observe that the optimal query plan never performs the merge join after the progressive join. If we first process query paths with the highest selectivity and the

intermediate result becomes rather small, then the progressive join becomes optimal. This leads to a simplification of the join selection algorithm, where we just split the query paths into two halves. The first half of query paths (possibly containing all query paths) is processed by a merge join algorithm and the second half of query paths is then finished by a progressive join.

6.2 Future Work

The thesis describes efficient approaches to twig pattern query searching, however, the problem of XQuery and XPath processing is more complex. We would like to do more work toward a full fledged XQuery processor. Mainly the processing of nested queries and queries with an node order among siblings condition would be particularly important. We understand the problem of an XML indexing and query processing as a similar problem to relational database indexing and SQL query processing. We do not need always index all parts and values of an XML document and we should try to preserve updatability of an XML database. Therefore, we try to develop a solution, where a database specialist can choose from a range of indexing possibilities which can support his query processing needs.

We would like to consider also optimality of other holistic approaches like the GeneralTwigStackList [25] in the context of recursive XML documents. We can expect that the recursiveness of some labeled path can be handled a with higher space complexity of the holistic algorithm.

There is also an update consideration. We should be always able to update our indexing scheme efficiently. If we are using the Dewey order or OrdPath labeling scheme, some compression can be used. This can considerably reduce the index size and the query processing can be positively influenced.

Author's Bibliography

[B1] R. Bača, and M. Krátký. On the Efficient Search of an XML Twig Query in Large DataGuide Trees. *Submited in Twenty Fifth International Conference on Data Engineering, ICDE 2009*, IEEE CS, 2009.

[B2] R. Bača, and M. Krátký, and V. Snášel. On the Efficient Search of an XML Twig Query in Large DataGuide Trees. In *Proceedings of the Twelfth International Database Engineering & Applications Symposium, IDEAS 2008*, Acceptance Rate 37%, IEEE, 2008.

[B3] R. Bača, and M. Krátký. A Cost-based Join Selection for XML Twig Content-based Queries. In *Proceedings of the Third International Workshop on Database Technologies for Handling XML Information on the Web, DataX 2008*, Acceptance Rate 26%, ACM DL, 2008.

[B4] M. Krátký, and R. Bača, and V. Snášel. Implementation of XPath Axes in the Multi-dimensional Approach to Indexing XML Data. In *Proceedings of the 18th International Conference on Database and Expert Systems Applications, DEXA 2007*, volume LNCS 4653/2007, pages 1–12, Acceptance Rate 27%, Springer–Verlag, 2007.

[B5] R. Bača, and M. Krátký. On the Efficient Implementation of Context Keyword-based Querying for XML Data. In *Proceeding of the IADIS International Conference WWW/Internet 2007*, volume 2, pages 187–192, Portugal, 2007.

Local conferences and technical reports:

[B6] R. Bača, and V. Snášel, and J.Platoš, and M. Krátký, and E. El-Qawasmeh. The Fast Fibonacci Decompression Algorithm. Technical Report. arXiv:0712.0811v2, `http://arxiv.org/abs/0712.0811`, 2007

[B7] R. Bača, and M. Krátký. Indexing XML Data - The State of the Art. In *Proceedings of ITAT conference*, Slovakia, 2007.

[B8] R. Bača, and M. Krátký, and V. Snášel. Evaluation of Multidimensional Approach to Indexing XML data with Compressed R-trees. In *Proceedings of the ZNALOSTI conference*, pages 234–245, Czech republic, 2007.

[B9] R. Bača, and M. Krátký. A Compression Scheme for the R-tree Data Structure. In *Proceedings of ITAT conference*, pages 69–74, Slovakia, 2006.

[B10] R. Bača, and M. Krátký. A Comparison of Element-based and Path-based Approaches to Indexing XML Data. In *CEUR Workshop Proceedings, DATESO 2006*, pages 103–115, Czech republic, 2006.

Bibliography

[1] S. Al-Khalifa, H. V. Jagadish, and N. Koudas. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of International Conference on Data Engineering, ICDE 2002*. IEEE Computer Society, 2002.

[2] M. ARENAS and L. LIBKIN. A Normal Form for XML Documents. *ACM Transactions on Database Systems, TODS*, 29(1):195–232, 2004.

[3] R. Bayer. The Universal B-Tree for Multidimensional Indexing: General Concepts. In *Proceedings of WWCA'97, Tsukuba, Japan*. Springer–Verlag, 1997.

[4] D. Bitton and J. Gray. Disk Shadowing. In *Proceedings of the 14th International Conference on Very Large Data Bases, VLDB 1988*, pages 331–338. Morgan Kaufmann Publishers Inc., 1988.

[5] N. Bruno, D. Srivastava, and N. Koudas. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD 2002*, pages 310–321. ACM Press, 2002.

[6] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about Keys for XML. *Information Systems*, 28(8):1037–1063, 2003.

[7] B. Catania, A. Maddalena, and A. Vakali. XML Document Indexes: A Classification. *IEEE Internet Computing*, 9(5):64–71, 2005.

[8] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Twig2Stack: Bottom-up Processing of Generalized-tree-pattern Queries Over XML documents. In *Proceedings of International Conference on Very Large Databases, VLDB 2006*, pages 283–294. VLDB Endowment, 2006.

[9] T. Chen, J. Lu, and T. Ling. On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques. *Proceedings of the ACM International Conference on Management of Data, SIGMOD 2005*, pages 455–466, 2005.

[10] Y. Chen, S. B. Davidson, and Y. Zheng. BLAS: an Efficient XPath Processing System. In *Proceedings of ACM SIGMOD 2004*, pages 47–58, New York, NY, USA, 2004.

[11] Z. Chen, G. Korn, F. Koudas, N. Shanmugasundaram, and J. Srivastava. Index Structures for Matching XML Twigs Using Relational Query Processors. In *Proceedings of 13th International Conference on Data Engineering, ICDE 2005*, pages 1273–1273. IEEE CS, 2005.

[12] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of 1999 ACM SIGMOD International Conference on Management of Data*, pages 431–442. ACM Press, 1999.

[13] Ecma International. Office Open XML File Formats, OOXML, Dec 2006, `http://www.ecma-international.org/`.

[14] W. Fan and L. Libkin. On XML Integrity Constraints in the Presence of DTDs. *Journal of the ACM (JACM)*, 49(3):368–406, 2002.

[15] W. Fan and J. Siméon. Integrity constraints for XML. *Journal of Computer and System Sciences*, 66(1):254–291, 2003.

[16] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[17] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proceedings of the 28th international conference on Very Large Data Bases, VLDB 2002*, pages 95–106. VLDB Endowment, 2002.

[18] G. Gottlob, C. Koch, and R. Pichler. The Complexity of XPath Query Evaluation. In *Proceedings of ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 179–190. ACM Press New York, NY, USA, 2003.

[19] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *ACM Transactions on Database Systems (TODS)*, volume 30, pages 444–491. ACM Press, 2005.

[20] G. Gou and R. Chirkova. Efficiently Querying Large XML Data Repositories: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1381–1403, 2007.

[21] T. Grust. Accelerating XPath Location Steps. In *Proceedings of the 2002 ACM SIGMOD, Madison, USA*. ACM Press, June 4-6, 2002.

[22] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch Its (Axis) Steps. In *Proceedings of the 29th, International Conference on Very Large Databases, VLDB 2003*, pages 524–535. VLDB Endowment, 2003.

[23] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM SIGMOD 1984, Boston, USA*, pages 47–57, June 1984.

[24] T. Härder, M. Haustein, C. Mathis, and M. Wagner. Node Labeling Schemes for Dynamic XML Documents Reconsidered. *Data & Knowledge Engineering*, 60(1):126–149, 2007.

[25] L. Jiaheng. *Efficient Processing of XML Twig Pattern Matching, Ph.D. thesis*. National University of Singapore, August 2006.

[26] H. Jiang, H. Lu, and W. 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. ACM press, 2004.

[27] H. Jiang, H. Lu, W. Wang, and B. Ooi. XR-Tree: Indexing XML Data for Efficient. In *Proceedings of ICDE, 2003, India*. IEEE, 2003.

[28] H. Jiang, W. Wang, H. Lu, and J. Yu. Holistic twig joins on Indexed XML Documents. In *Proceedings of the 29th International Conference on Very Large Databases, VLDB 2003*, pages 273–284. VLDB Endowment, 2003.

[29] M. Krátký, J. Pokorný, and V. Snášel. Indexing XML Data with UB-trees. In *Proceedings of ADBIS 2002, Bratislava, Slovakia*, pages 155–164, 2002.

[30] M. Krátký, J. Pokorný, and V. Snášel. Implementation of XPath Axes in the Multi-dimensional Approach to Indexing XML Data. In *Current Trends in Database Technology, EDBT 2004*, volume LNCS 3268/2004. Springer–Verlag, 2004.

[31] M. Krátký, T. Skopal, and V. Snášel. Multidimensional Term Indexing for Efficient Processing of Complex Queries. *Kybernetika, Journal*, 40(3):381–396, 2004.

[32] M. Krátký, V. Snášel, P. Zezula, and J. Pokorný. Efficient Processing of Narrow Range Queries in the R-Tree. In *Proceedings of IDEAS 2006*. IEEE CS Press, 2006.

[33] H. Li, M. Lee, W. Hsu, and C. Chen. An Evaluation of XML Indexes for Structural Join. *Proceedings of the ACM International Conference on Management of Data, SIGMOD 2004*, 33(3):28–33, 2004.

[34] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 27th International Conference on Very Large Databases, VLDB 2001*, 2001.

[35] J. Lu, T. Chen, and T. Ling. Efficient Processing of XML Twig Patterns with Parent Child Edges: a Look-ahead Approach. In *Proceedings of the Thirteenth ACM conference on Information and knowledge management, CIKM 2004*, pages 533–542. ACM Press, 2004.

[36] J. Lu, T. Ling, C. Chan, and T. Chen. From Region Encoding to Extended Dewey: on Efficient Processing of XML Twig Pattern Matching. *Proceedings of the 31st International Conference on Very Large Databases, VLDB 2005*, pages 193–204, 2005.

[37] J. Lu, T. Ling, C. Chan, and T. Chen. From Region Encoding to Extended Dewey: on Efficient Processing of XML Twig Pattern Matching. In *Technical report*. TRA6/05 National university of Singapore, 2005.

[38] T. S. M. Yoshikawa, T.Amagasa and S. Uemura. XRel: a Path-based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Transactions on Internet Technology*, 1(1):110–141, 2001.

[39] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *ACM SIGMOD Record*, 26(3):54–66, 1997.

[40] J. McHugh and J. Widom. Query Optimization for XML. In *Proceedings of the 25th International Conference on Very Large Data Bases, VLDB 1999*, pages 315–326. Morgan Kaufmann Publishers Inc., 1999.

[41] Michael Kay. Saxon: The XSLT and XQuery processor. `http://www.saxonica.com/`.

[42] M. Moro, Z. Vagena, and V. Tsotras. Tree-pattern Queries on a Lightweight XML Processor. In *Proceedings of the 31st international conference on Very large data bases*, pages 205–216. VLDB Endowment, 2005.

[43] Oasis. OpenDocument format v1.0, OASIS Standard, 1 May 2006, `http://www.oasis-open.org/`.

[44] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML Node Labels. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 903–908. ACM press, 2004.

[45] J. Pokorný. *XML: a Challenge for Databases?*, pages 147–164. Kluwer Academic Publishers, Boston, 2001.

[46] N. Polyzotis and M. Garofalakis. Structure and Value Synopses for XML Data Graphs. In *Proceedings of International Conference on Very Large Databases, VLDB 2002*. Morgan Kaufmann, 2002.

[47] N. Polyzotis and M. Garofalakis. XSKETCH Synopses for XML Data Graphs. *ACM Trans. Database Syst.*, 31(3):1014–1063, 2006.

[48] R. Cover. XML Applications and Initiatives, 2005, `http://xml.coverpages.org/xmlApplications.html`.

[49] J. W. R. Goldman. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of VLDB*, pages 436–445, 1997.

[50] P. Rao and B. Moon. PRIX: Indexing and Querying XML Using Prufer Sequences. In *Proceedings of 20th International Conference on Data Engineering, ICDE 2004*, pages 288–299. IEEE, 2004.

[51] J. Robie. XML Query Language (XQL), 1999, `http://metalab.unc.edu/xql/xql-proposal.xml`.

[52] Roger L. Costello. XML Schemas: Best Practices, 2001, `http://www.xfront.com/BestPracticesHomepage.html`.

[53] A. R. Schmidt and at al. The XML Benchmark. Technical Report INS-R0103, CWI, The Netherlands, April, 2001, `http://monetdb.cwi.nl/xml/`.

[54] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of the 25th International Conference on Very Large Databases, VLDB 1999. Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.

[55] S. S.Prakas and S.Madria. SUCXENT: An Efficient Path-Based Approach to Store and Query XML Documents. In *Proceedings of Database and Expert Systems Applications, DEXA 2004*, volume LNCS 3180/2004, pages 285–295. Springer-Verlag, 2004.

[56] I. Tatarinov and at al. Storing and Querying Ordered XML Using a Relational Database System. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD 2002*, pages 204–215, New York, USA, 2002. ACM Press.

[57] Database Research Group at the University of Michigan. TIMBER: Tree-structured native XML database Implemented at the University of Michigan by Bright Energetic Researchers, 2001, `http://www.cis.upenn.edu/~treebank/`.

[58] University of Pennsylvania. The Penn Treebank Project, 2001, `http://www.eecs.umich.edu/db/timber/`.

[59] W3 Consortium. Extensible Markup Language (XML) 1.0, W3C Recommendation, 10 February 1998, `http://www.w3.org /TR/REC-xml`.

[60] W3 Consortium. XQuery 1.0: An XML Query Language, W3C Working Draft, 12 November 2003, `http://www.w3.org/TR/xquery/`.

[61] W3 Consortium. XQuery Update Facility 1.0, W3C Candidate Recommendation, 14 March 2008, `http://www.w3.org/TR/xquery-update-10/`.

[62] W3 Consortium. XML Path Language (XPath) Version 2.0, W3C Working Draft, 15 November 2002, `http://www.w3.org/TR/xpath20/`.

[63] W3 Consortium. XML Pointer Language (XPointer) Version 1.0, W3C Working Draft, 16 August 2002, `http://www.w3.org/TR/xptr/`.

[64] W3 Consortium. XML Schema Part 1: Structure, W3C Recommendation, 2 May 2001, `http://www.w3.org/TR/xmlschema-1/`.

[65] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: a Dynamic Index Method for Querying XML Data by Tree Structures. In *Proceedings of the ACM SIGMOD 2003*, pages 110–121. ACM Press, 2003.

[66] I. Witten, A. Moffat, and T. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images. *Information Theory, IEEE Transactions on*, 41(6), 1995.

[67] Y. Wu, J. Patel, and H. Jagadish. Estimating Answer Sizes for XML Queries. In *Proceedings of Advances in Database Technology – EDBT 2002*, LNCS, Volume 2287/2002. Springer-Verlag, 2002.

[68] Y. Wu, J. M. Patel, and H. Jagadish. Structural Join Order Selection for XML Query Optimization. In *Proceedings of the 19th International Conference on Data Engineering, ICDE 2003*, pages 443 – 454. IEEE Computer Society, 2003.

[69] University of Washington's Database Group. The XML Data Repository, 2002, `http://www.cs.washington.edu/research/xmldatasets/`.

[70] T. Yu, T. Ling, and J. Lu. TwigStackList¬: A Holistic Twig Join Algorithm for Twig Query with Not-predicates on XML Data. In *Database Systems for Advanced Applications: 11th International Conference, DASFAA 2006*, volume 3882 of *LNCS*, pages 249–263. Springer-Verlag New York Inc., 2006.

[71] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD 2001*, pages 425–436, New York, USA, 2001. ACM Press.