EFFICIENT PROCESSING OF REGULAR QUERIES OVER XML DATA SETS USING STRUCTURAL INDEXES

Vu Le Anh

PhD. Dissertation

Supervisor:
Dr. Attila Kiss

Budapest, Hungary
December 2007
# Table of Contents

Table of Contents \hspace{1cm} v

List of Tables \hspace{1cm} vi

List of Figures \hspace{1cm} vii

List of Abbreviations \hspace{1cm} viii

Acknowledgements \hspace{1cm} ix

1 General Introduction \hspace{1cm} 1

2 Data Graph, Regular Queries And Naive Processing Algorithms \hspace{1cm} 5
   2.1 Data Graph, Query Graph \hspace{1cm} 6
   2.1.1 Data Graph \hspace{1cm} 6
   2.1.2 Query Graph \hspace{1cm} 8
   2.2 Naive Graph Traversal Algorithm \hspace{1cm} 11
   2.3 Naive Joining Graph Algorithm \hspace{1cm} 13
      2.3.1 Edge-Labeled Data Graph \hspace{1cm} 13
      2.3.2 State Data Graph \hspace{1cm} 14
      2.3.3 Naive Joining Graph Algorithm \hspace{1cm} 16
   2.4 Classification, Equivalence And Decomposition Of Regular Queries \hspace{1cm} 19
      2.4.1 Classification And Equivalence Of Regular Queries \hspace{1cm} 19
      2.4.2 Decomposition Of Regular Queries \hspace{1cm} 21
   2.5 Summary \hspace{1cm} 22

3 Structural Indexes, DL-Indexes \hspace{1cm} 24
   3.1 Preliminaries \hspace{1cm} 26
      3.1.1 Near Primary Queries And Simple Alternation Path Expressions \hspace{1cm} 26
      3.1.2 Structural Indexes \hspace{1cm} 27
   3.2 Partitions And Split Operation \hspace{1cm} 30
      3.2.1 Partitions \hspace{1cm} 30
<table>
<thead>
<tr>
<th>3.2.2</th>
<th>Split Operation</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>1-Index And DL-1-Index</td>
<td>33</td>
</tr>
<tr>
<td>3.3.1</td>
<td>1-Index And Stable Structural Indexes</td>
<td>33</td>
</tr>
<tr>
<td>3.3.2</td>
<td>DL-1-index</td>
<td>34</td>
</tr>
<tr>
<td>3.4</td>
<td>DL-A*(k)-Index</td>
<td>38</td>
</tr>
<tr>
<td>3.4.1</td>
<td>A(k)-Index And k-Stable Structural Indexes</td>
<td>38</td>
</tr>
<tr>
<td>3.4.2</td>
<td>DL-A*(k)-Index</td>
<td>40</td>
</tr>
<tr>
<td>3.5</td>
<td>Summary</td>
<td>44</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>Efficient Processing Of Primary Queries Over Near Tree XML Data Sets</th>
<th>46</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Preliminaries</td>
<td>47</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Tree Structure Indexes</td>
<td>48</td>
</tr>
<tr>
<td>4.2</td>
<td>Link Graph And Improved Algorithm</td>
<td>50</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Link Graph</td>
<td>50</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Complex Representation Of Primary Queries</td>
<td>52</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Improved Algorithm</td>
<td>53</td>
</tr>
<tr>
<td>4.3</td>
<td>Strong 1-Index</td>
<td>55</td>
</tr>
<tr>
<td>4.4</td>
<td>Index Constructions</td>
<td>58</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Tree Structure Index Construction</td>
<td>58</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Link Graph Construction</td>
<td>59</td>
</tr>
<tr>
<td>4.5</td>
<td>Summary</td>
<td>62</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>Efficient Processing Of Regular Queries Over Fragmented XML Tree</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Preliminaries</td>
<td>66</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Fragmented Tree</td>
<td>66</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Processing Over Fragment</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>Stream- And Partial Processing Approaches</td>
<td>68</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Tree Traversal Algorithm</td>
<td>68</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Partial Parallel Processing Algorithm</td>
<td>70</td>
</tr>
<tr>
<td>5.3</td>
<td>Efficient Processing Regular Queries Using Tree And Structural Indexes</td>
<td>72</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Tree Indexes</td>
<td>73</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Structural Indexes</td>
<td>74</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Processing Queries With Tree- And Structural Indexes</td>
<td>76</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary</td>
<td>77</td>
</tr>
</tbody>
</table>

| 6     | Summary Of The Dissertation | 79 |

Appendixes 82
A Experiments For DL-Indexes
  A.1 DL-1-Index vs. 1-Index ........................................ 84
  A.2 DL-A*\textsuperscript{(k)}-Index vs. A(k)-Index ..................... 84

B Experiments For NA-, NI-, NT- And TI Algorithms ................. 86
  B.3 Performance Result .............................................. 87

C Experiments For Different Regular Query Processing Algorithms Over Fragmented XML Tree ........................................ 89

Bibliography ........................................................... 92
List of Tables

A.1 The properties of the data sets using in the experiments of DL-indexes. . . 82

B.1 The properties of the data sets using in the experiments of processing
   algorithms over near tree XML data sets. . . . . . . . . . . . . . . . . . . . 87
B.2 The properties of the 1-index and the strong 1-index using in the experi-
   ments of processing algorithms over near tree XML data sets. . . . . . . . 87
B.3 The average of the costs of the queries evaluation with N-, NI-, NT- and
   TI algorithms. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
C.1 The results of the experiments of processing algorithms over fragmented
   XML tree. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
List of Figures

2.1 An example of data graph. .................................................. 7
2.2 Query graph of $R = a.(b|c)^* . a.$ ........................................ 10
2.3 Node-labeled and edge-labeled graphs representing same XML data set with $\Sigma = \{a, b, c\}$. .................................................. 14
2.4 State data graph of the edge-labeled data graph shown in Figure 2.3 and the query graph shown in Figure 2.2. ................................. 15
2.5 An example of the NAIVE-JOINING algorithm. ....................... 17
3.1 Query graph of the SAPE $S_0S_1 \ldots S_l$. ............................... 27
3.2 An example of $DL\text{-}I$-indexes ............................................. 36
3.3 An example of the $DL\text{-}A^*/(1)$-index .................................. 42
4.1 An example of tree structure index ......................................... 49
4.2 Link graph of the data graph shown in Figure 4.1 ..................... 51
4.3 An example of the strong 1-index ......................................... 57
5.1 The model of share-nothing parallel database systems. ............... 64
5.2 An example of fragmented XML tree and regular query ............... 67
5.3 Tree and structural indexes of the fragmented tree in Figure 5.2(b) . 73
A.1 The results of the experiments ............................................. 83
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL</td>
<td>Dynamic Labeling</td>
</tr>
<tr>
<td>SAPE</td>
<td>Simple Alternation Path Expression</td>
</tr>
<tr>
<td>SHPDS</td>
<td>Shared-Nothing Parallel Database System</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>NA</td>
<td>Naive Algorithm</td>
</tr>
<tr>
<td>NI</td>
<td>Naive algorithm over the 1-Index</td>
</tr>
<tr>
<td>NT</td>
<td>Improved algorithm using tree structure index over the data graph</td>
</tr>
<tr>
<td>TI</td>
<td>Improved algorithm using tree structure index over the strong 1-index</td>
</tr>
</tbody>
</table>
Acknowledgements

First and foremost, I wish to express my profound gratitude to my supervisor, Dr. Attila Kiss, for his thoughtful support and careful guidance at every stage of the development of this research. Also, I would like to thank Prof. András Benczúr, the head of the Department of Information Systems and the Information Systems Doctoral Program. Without the opportunities of doctoral and pre-doctoral scholarships, I could not have the prerequisite conditions to accomplish this work.

Furthermore, I am extremely grateful to all of my colleagues from the Department of Information Systems and the e-Science Regional Knowledge Center ELTE, the unknown reviewers of papers for their discussions, valuable comments, and all other means of support.

I also would like to thank Dr. Dung Luong Dinh and Huyen Anh Nguyen for their careful proof-reading of my thesis. Last but not least, my heartfelt thanks, as always, to my beloved parents, my two younger brothers, Kien Le Trung and Hieu Le Trung, and my friends whose great care and encouragement keep me highly motivated for the fulfilment of this dissertation in particular and the doctoral course in Hungary in general.
Chapter 1

General Introduction

In recent years, eXtensible Markup Language (XML) has emerged as a new standard for representing, exchanging and querying data on the Internet. XML structure provides us two simple, but effective underlying ideas. The first one is the using of the tags on data elements which not only specify how the data should be formatted but also identify the meaning of the data. The second one is the expression of the relationships between data elements by using simple nesting and reference edges. XML offers to its users many advantages especially due to its simpleness, easy parsing, self-describing, extendibility and inter-operability. Thus, XML is a good choice for representing various types, dynamic schema or open data sets such as scientific databases, clustered web storages, opening directories, to name a few.

In this thesis, the XML data sets under the consideration are assumed to be very large and integrated by data management systems. They have various types of structure and storage. The structure of these data sets can be tree, near-tree or general graph. An XML is a tree if the reference edges are not considered. It is a near tree if the number of reference edges is much smaller than the number of edges. An XML can be stored in a machine or distributed over the internet.

Several query languages such as XPath [1], XML-QL [7], UnQL [5], XQuery [2] have been proposed to retrieve XML and semi-structured data. The common feature of these languages is the usage of regular path expressions (or regular queries). Normally, the result of a regular path expression is defined as the set of data nodes in which there is a path from the root to each node matching this expression. In general case, the result of a regular query is a set of pairs of nodes which is limited by an input set and an output
set. The first node of a pair and the second one are an element of the input set and the output set respectively. Furthermore, there is a path from the first node to the second one matching the regular path expression. This thesis focuses on the regular queries. The most important classes of the regular queries: primary-, near primary- and simple alternation path expression (SAPE for short) queries are also mentioned.

The structural indexes (1-index \cite{18}, A(k)-index \cite{12}, D(k)-index \cite{6}, M*(k)-index \cite{11}, etc) have been proposed to speed up the query processing over XML data. The basic idea is that the data graph is simulated by an index graph. The size of the index graph is often much smaller than that of the data graph. The index graph is equivalent to the data graph over a given set of queries. It means that the results of these queries over the index graph coincide with those of the same queries over the data graph. As the size of the index graph is often much smaller than that of the data graph, the cost and the waiting time of processing queries over the index graph is often much lower than that over the data graph.

Based on our results \cite{P1,P2,P3,P5,P6} and the most popular algorithms, structural indexes, I analyze several efficient regular query processing algorithms over XML data sets using structural indexes in different contexts. The decomposition and the equivalence of the regular queries are also investigated. The following issues will be discussed in the thesis.

Chapter 2 gives an overview of the processing of regular queries over very large XML data sets. The graph models for XML data sets and regular queries are introduced. We study two simple processing algorithms: graph traversal algorithm and "joining" graph algorithm. The data graph and queries, in the "joining" graph algorithm, are represented in relations and can be processed in relational databases. The corresponding query processing algorithm is written in relational algebra and Datalog \footnote{Datalog is a query and rule language for deductive databases that syntactically is a subset of Prolog.}. The decomposition and the equivalence of regular queries are also investigated. It is shown that each regular query can be decomposed into primary queries.

Chapter 3 is about structural indexes and the DL-indexes. A mathematical background for structural indexes is introduced and studied. Based on the split-, refine operations of partitions, the DL-indexes that support dynamic labeling are studied (DL is the...
abbreviation of dynamic labeling). In most of current structural indexes, the data nodes in a same index node must have the same label. These indexes are called static labeling structural indexes. The static labeling structural indexes do not support the queries containing alternative or wild card well. Dynamic labeling techniques permit the data nodes in the same index node to have different labels, which make the index has smaller size and better support for SAPE queries. Each class of static labeling structural indexes can be developed to be a new class of DL-indexes. DL-1-index and DL-A*(k)-index which are developed from the 1-index and the A(k)-index are investigated.

Chapter 4 is about processing primary queries over near-tree XML data sets. The number of reference edges of a near tree data graph is much smaller than that of edges. Data graph is decomposed into two components: basic tree and link graph. The basic tree is indexed so that the processing cost over it is very low. Since the data graph is a near tree, the link graph is very small comparing with the data graph. The primary query processing algorithm traverses the link graph and processes over the tree structure index of the basic tree. Moreover, a new algorithm combining two techniques: structural- and tree structure indexes is also introduced. The data graph is simulated by the strong 1-index, in which the basic tree structure is conserved. The new algorithm processes primary queries over the tree structure index and the link graph of the strong 1-index. Efficient algorithms for building the tree structure indexes and the link graph in linear complexity are proposed.

Chapter 5 is about the processing of regular queries over the very large XML trees which are fragmented over a shared-nothing parallel database system \[8,23,24\] (SNPDS for short). SNPDS is a system of machines connected to each other by high-speed links. The fragments which are subtrees of XML tree are stored on machines. The partial processing approach is the most popular and famous algorithm for processing regular queries on SNPDS. The query is sent to and evaluated partially at each machine. The parallelism reduces the waiting time, but there are many unnecessary operations to be computed. Our algorithm is also based on the partial evaluation, but unnecessary operations are restrained by tree index and structural indexes. Two types of unnecessary fragment operations are defined. The unnecessary fragment operations type 1 are unreachable operations that are determined by preprocessing on the tree index. The unnecessary operations type 2
return no result. These are restrained by preprocessing on the structural indexes of the fragments. Since the tree- and the structural indexes are small, our algorithm overcomes the existing algorithms according to criteria of the waiting time or the total processing and communication cost.

The thesis is organized as follows. Chapter 2 gives an overview of the processing of regular queries over very large XML data sets. Chapter 3 provides the mathematical foundation of structural indexes and DL-indexes. The efficient primary query processing algorithms over near-tree XML data sets are proposed in Chapter 4. Chapter 5 investigates the efficient regular query processing algorithm on fragmented XML trees. Finally, Chapter 6 concludes the thesis. In addition, appendixes describe the experiment results in [P2, P3, P5]. Appendix A introduced in [P3] is the experiment of the performance of DL-indexes versus classical structural indexes. Appendix B originated from [P2] is the performance evaluation of different processing algorithms of primary queries over near-tree XML data sets. Appendix C mentioned in [P5] is the performance of different processing algorithms processing regular queries over fragmented XML trees.
Chapter 2

Data Graph, Regular Queries And Naive Processing Algorithms

In all of works [3, 5, 7, 10, 15, 16, 18] about the processing queries over XML, both of data sets and regular queries are represented by rooted and labeled graphs. The difference of these labeled graphs is that only nodes, or only edges or both of them are tagged by label values. However, the nature of those models are the same and we can convert one graph model to the others [18]. In this work, the node labeled graph is proposed as the original graph model of XML data sets, and the edge labeled graph is proposed as the graph model of regular queries. The graphs represented data set and regular query are called data graph and query graph, respectively. The result of a query over a data set is based on the matching of data node path and query node path.

This chapter gives an overview of the processing of regular queries over very large XML data sets. The result of a regular query over an XML data set can be computed by two naive algorithms. The first one is the graph traversal algorithm, which is used in many works about the structural indexes [6, 11, 12]. Both of the query graph and the data graph are traversed together to find all data node paths matching some query node path. The second one is the joining graph algorithm, which is introduced in our paper [P1]. The data set and the queries are represented by relations and the processing algorithm is written in relational algebra and Datalog. With this purpose, we would like to introduce an alternative graph model for XML data graph. The result is determined by the transitive closed relation of the state-data graph which is the result of joining the data graph and the query graph. XML data set, regular queries and the processing
algorithm can be represented, stored and processed in relational database systems. The
equivalence and the decomposition of regular queries are discussed at last. We will show
that each regular query is equivalent to the union of a finite set of primary queries.

This chapter is organized as follows. First, the graph models for XML data sets and
regular queries are introduced in Section 2.1. Then, Section 2.2 represents the graph
traversal algorithm. Section 2.3 introduces the ”joining” graph algorithm. The equiv-
alence and the decomposition of regular queries are discussed in Section 2.4. Finally,
Section 2.5 concludes the chapter.

2.1 Data Graph, Query Graph

2.1.1 Data Graph

Definition 2.1. (Data graph) The data graph of an XML data set is a rooted directed
node-labeled graph
\[ G = (V, E, \Sigma, r, label) \].

Here \( V \) is the set of nodes. \( E = E_b \cup E_f \) is the set of edges representing relationships
between nodes. \( E_b \) is the set of basic edges representing element-subelement, element-
attribute or element-value relationship; \( E_f \) is the set of reference edges representing ID/IDREF
relationship. \( \Sigma \) is the set of label values. \( label: V \mapsto \Sigma \) is the node-labeling function. Each
node \( u \) is labeled by a \( label(u) \in \Sigma \). \( r \) is the single root of the graph with no incoming
edges.

\( T = (V, E_b) \) is a tree. \( T \) is called the basic tree of the data graph.

Figure 2.1 shows the data graph of the following XML document which represents the
PhD students, the professors of a Computer Science Department and their publications:

\[
\begin{align*}
\text{CSDepartment} & \\
\text{PhDStudents} & \\
\text{Student id="s1"} & \\
\text{Name} & \text{John} \\
\text{Papers} & \\
\text{Paper id="pp1"} & \\
\text{Title} & \text{ABC} \\
\text{Author} & \text{Dr.Ben} \\
\text{Author idref="s1"} & \\
\text{Paper} & \\
\text{Papers} & \\
\text{Student id="s2"} & \\
\text{Name} & \text{Tom} \\
\end{align*}
\]
Figure 2.1: An example of data graph.
In Figure 2.1, XML nodes of the document are represented by data nodes of the graph. The basic edges are solid, and the reference edges created by ID/IDREF relationships are dotted. Each data node is labeled by their tag name or attribute name in case of element or attribute node, and by text value in case of text node. The set of text nodes is \{5,7,11,13,15,19,21,25,27,32,35,37,41\}. The root of the data graph is node 1.

In reality e.g. Lore database system [16, 17], the text nodes are stored separately in hash tables. It enables us to find quickly those nodes which satisfy arithmetic or string-pattern conditions. Moreover, most classical structural indexes just represent the data graphs without text nodes. In this case, \(\Sigma\), the set of tag names and attribute names, is small and finite. In the rest of the thesis, we do not consider the text nodes. As a result, \(\Sigma\) is finite, and \(|\Sigma|\) is considered as a constant.

Let \(u_1, u_2, \ldots, u_k\) be data nodes. \(p = u_1u_2\ldots u_k\) is a data node path if \((u_i, u_{i+1})\) is an edge \((i = 1, \ldots, k - 1)\). The length of \(p\), denoted by \(\text{length}(p)\), is \(k\). The label of \(p\) denoted by \(\text{label}(p) \in \Sigma^*\) is \(\text{label}(u_0).\text{label}(u_1)\ldots\text{label}(u_k)\). In the data graph shown in Figure 2.1 the length of the data node path \(p = 9\).16.23.26\) is 4, and \(\text{label}(p) = \text{paper.author.professor.name}\).

Let \(u \in V\) be a data node, \(W \subseteq V\) be a set of data nodes, and \(S \subseteq \Sigma\) be a set of label values. Then, we define:

\[
\text{Succ}(u) := \{v \in V | (u, v) \in E\}, \quad \text{Succ}(W) := \bigcup_{u \in W} \text{Succ}(u),
\]

\[
\text{Succ}_b(u) := \{v | (u, v) \in E_b\}, \quad \text{Succ}_b(W) := \bigcup_{u \in W} \text{Succ}(u),
\]

\[
\text{Succ}_f(u) := \{v | (u, v) \in E_f\}, \quad \text{Succ}_f(W) := \bigcup_{u \in W} \text{Succ}(u),
\]

\[
\text{label}(W) := \bigcup_{u \in W} \{\text{label}(u)\}, \quad \text{label}^{-1}(S) := \{u \in V | \text{label}(u) \in S\}.
\]

2.1.2 Query Graph

The foundations, properties and applications of automata, regular languages and regular path expressions can be founded in [21]. In this section, only the definitions and properties of regular path expressions used in the thesis are introduced.
The regular queries of the data graph is given by a regular path expression over \( \Sigma \), the set of labels. The syntax of regular path expressions over alphabet \( \Sigma \) is:

\[
R = \varepsilon \mid \_ \mid a \mid R_1.R_2 \mid R_1|R_2 \mid R_1^*
\]

where \( \varepsilon \) is the empty word; \( \_ \) is an arbitrary letter; \( a \in \Sigma \) is a letter; \( R_1, R_2, R \) are regular path expressions; \( (.), (|), (*) \) are the concatenation-, alternation-, and iteration operations, respectively.

Each regular path expression \( R \) over \( \Sigma \) denotes a regular language \( L(R) \) over \( \Sigma \) according to the following conventions:

1. \( L(\varepsilon) \) is the empty language.
2. \( L(\_) := \Sigma \) and \( \forall a \in \Sigma : L(a) := \{a\} \).
3. \( \forall R_1, R_2 \) regular expressions:

\[
L(R_1|R_2) := L(R_1) \cup L(R_2),
L(R_1.R_2) := L(R_1).L(R_2),
L(R_1^*) := L(R_1)^*.
\]

\( (.), (\cup), (*) \) are the concatenation-, union-, and iteration operations upon languages respectively.

**Definition 2.2.** (Pair of data nodes matching regular query) An ordered pair of data nodes \((u, v)\) of data graph \( G \) matches regular expression \( R \) (denoted by \( u \xrightarrow{R\ G} v \)) if there exists a data node path \( p = u_0u_1 \ldots u_k \) from \( u \) to \( v \) \((u_0 = u, u_k = v)\) such that \( \text{label}(p) \in L(R) \).

**Definition 2.3.** (The result of a regular query over data graph) The natural result of regular path expression \( R \) on data graph \( G \) is the set of data nodes to which there is a data node path matching \( R \) from the root.

\[
\mathcal{R}(G, R) := \{v \in V | r \xrightarrow{R\ G} v \}.
\]

The general result of regular path expression \( R \) on data graph \( G \) with input set \( I \subseteq V \), and output set \( O \subseteq V \) is:

\[
\mathcal{R}_I^O(G, R) := \{(u, v) \in I \times O | u \xrightarrow{R\ G} v \}.
\]
The natural result of regular path expression $R$ can be considered a special case of the general result of $R$ with the input set $I = \{r\}$ and output $O = V$. To make it more convenient, in the rest of the thesis we will use the phrase "the result" for the phrase "the natural result" and the phrase "the result of regular expression $R$ with input $I$ and output $O$" for the the phrase "the general result of regular expression $R$ with input $I$ and output $O$". The result of $R = .*$.Professor.*.Paper.Title ("Finding the title of the papers of the professors") over the data graph shown in Figure 2.1 is $\{12, 31, 40\}$.

Each regular expression can be computed by a non-deterministic finite automaton (NFA for short). An NFA can be reduced such that there is no $\epsilon$-transition. The transition graph defining the non $\epsilon$-transition NFA is called query graph of the query.

**Definition 2.4.** (Query graph) Let $R$ be a regular path expression, and $A$ be the non $\epsilon$-transition NFA computing $L(R)$. The transition graph of $A$, $Q_R = (V_R, E_R, \Sigma, s_0, F)$, is called the query graph of $R$. Here $V_R$ is the set of states. $\Sigma$ is an alphabet. $s_0 \in V_Q$ is the start state and also the root of query graph. $F \subseteq V_R$ is the set of final states. $E_R \subseteq V_R \times \Sigma \times V_R$ is the set of transition edges. $e = (a, l, b) \in E_R$ if and only if the automata changes from state $a$ to state $b$ in the case the input equals to $l$.

The algorithm for computing the transition graph and reducing epsilon transitions of a regular expressions can be found at [21].

Figure 2.2 illustrates an example of query graph. $s_0$ is the start state, $s_2$ is the final state and the regular expression is $R = a.(b|c)^*a$.

![Query graph of R = a.(b|c)^*a.](image)

Suppose the current state of the automata is $a$, and the input is $l \in \Sigma$. Then, we define $E_R(a, l)$ as the set of possible states to which the automata will change. Clearly,

$$E_R(a, l) = \{ b \mid (a, l, b) \in E_R \}.$$
Suppose the current state of the automata is $a$, and the input is a value of $S \subseteq \Sigma$. Then, we define $E_R(a, S)$ as the set of possible states to which the automata will change. Clearly,

$$E_R(a, S) = \bigcup_{l \in S} E_R(a, l).$$

Suppose $p = u_1 \ldots u_k$ is a data node path, and $p_q = q_0 \ldots q_k$ is a query node path. Then, we say that $p$ matches $p_q$ if

$$\forall i = 1, \ldots, k : q_i \in E_R(q_{i-1}, \text{label}(u_i)).$$

In this case, we also say that $(u_1, u_k)$ matches $(q_0, q_k)$.

**Corollary 2.1.** An ordered pair of data nodes $(u, v)$ of data graph $G$ matches regular expression $R$ if and only if $(u, v)$ matches $(s_0, f)$. Here, $s_0$ is the start state, and $f$ is a final state of the query graph of $R$.

**Proof.** Suppose $p = u_1 \ldots u_k$ is the data node path from $u$ to $v$, which matches the query node path $p_q = q_0 \ldots q_k$ from $s_0$ to $f$. Clearly, $q_{i+1}$ is a possible state to which the automata will change to in the case the current state is $q_i$, and the input is $\text{label}(u_{i+1})$. Hence, $q_k$ is a possible state to which the automata will change in the case the current state is $q_0$, and the input is $\text{label}(p)$. Since $q_0 = s_0$ and $q_k = f$, $\text{label}(p) \in L(R)$. Therefore, the corollary holds.

**Definition 2.5.** *(Distance of matching query nodes and data nodes)* Suppose $(u, v)$ matches $(s, q)$. Let $p$ be the shortest path connecting from $u$ to $v$ in which there exists a query node path $p_q$ from $s$ to $q$ such that $p$ matches $p_q$. Then, we define $\text{length}(p)$ as the distance between $(u, v)$ and $(s, q)$.

### 2.2 Naive Graph Traversal Algorithm

We traverse the data graph and the query graph in the tours to find the matches of data nodes and query nodes with following rules:

1. Each tour starts at some data node of the input set and the root of the query graph.
2. Each pair of data node and query node $(u, q)$ is visited maximum once in each tour.
3. If $(u, q)$ is visited, $(u, v)$ is a data edge and state $s$ is an element of $E_R(q, \text{label}(u))$, $(v, s)$ is also visited.
Formally, the result of regular query $R$ represented by query graph $Q$ over data graph $G$ with input set $I$ and output set $O$ is computed by the \textsc{Naive-Graph-Traversal} algorithm.

\begin{verbatim}
\textsc{Naive-Graph-Traversal}(G, Q, I, O)
begin
1. Result ← ∅
2. for each $u_0 \in I$ do
3.  $Visited ← ∅$
4.  $\text{Scan}(u_0, s_0)$
5. return Result
end
\end{verbatim}

\textsc{Naive-Graph-Traversal}(G, Q, I, O) \quad $\text{Scan}(u, q)$
begin
1. $Visited ← Visited \cup (u, q)$
2. for each state $s \in E_R(q, label(u))$ do
3.  if $s \in F \land u \in O$ then
4.  $Result = Result \cup \{(u_0, u)\}$
5. for each edge $(u, v) \land (v, s) \notin Visited$ do
6.  $\text{Scan}(v, s)$
end

\textbf{Lemma 2.1.} Suppose $u_0$ is an element of $I$, and $Visited(u_0)$ is the value of variable $Visited$ at the end of the tour beginning at $(u_0, s_0)$. Then, we have:

\[(u_0, u) \text{ matches } (s_0, s) \text{ if and only if there exists } (u, q) \in Visited(u_0) \text{ such that } s \in E_R(q, label(u)).\]

\textbf{Proof.} Suppose $(u_0, u)$ matches $(s_0, s)$, and $l$ is the distance between $(u_0, u)$ and $(s_0, s)$. We will prove the lemma using the inductive method by the value of $l$. If $l = 1$, we have: $(u_0, u_0)$ matches $(s_0, s) \iff s \in E_R(s_0, label(u_0))$.

Since the function $\text{Scan}(u_0, s_0)$ is executed, $(u_0, s_0) \in Visited(u_0)$. Hence, the lemma is true when $l = 1$.

Suppose $(u_0, u)$ matches $(s_0, s)$ and the distance between them is $l$ ($l > 1$). Let $p = u_0u_1 \ldots u_{l-1}$ be the data node path connecting from $u_0$ to $u$ ($u_{l-1} = u$), and $p_q = s_0s_1 \ldots s_l$ be the data node path connecting from $s_0$ to $s$ ($u_l = s$) in which $p$ matches $p_l$. Then, we have:

(i) $(u_0, u_{l-2})$ matches $(s_0, s_{l-1})$;
(ii) The distance between $(u_0, u_{l-2})$ and $(s_0, s_{l-1})$ is less than $l$;
(iii) $s \in E_R(s_{l-1}, label(u))$.

The inductive condition and (ii) implies that:

\[\exists q : (u_{l-2}, q) \in Visited(u_0) \land s_{l-1} \in E_R(q, label(u_{l-1})).\]

Since $(u_{l-2}, q) \in Visited(u_0)$, the function $\text{Scan}(u_{l-2}, q)$ must be processed. Since $(u_{l-2}, u)$ is an edge and $s_{l-1} \in E_R(q, label(u_{l-1}))$, the $\text{Scan}(u, s_{l-1})$ must be processed. Hence, $(u, s_{l-1}) \in Visited(u_0)$. Furthermore, $s \in E_R(s_{l-1}, label(u))$ so $(u, s_{l-1})$ satisfies the condition of the lemma. Therefore, the lemma holds. \qed

\textbf{Proposition 2.1.} The \textsc{Naive-Graph-Traversal}(G, Q, I, O) algorithm stops after finite steps and returns the result of regular query $R$ represented by query graph $Q$ over data graph $G$ with input set $I$ and output set $O$. The complexity of the algorithm is $O(|I|.m.k)$ in which $m$ is the number of edges of $G$, $k$ is the number of query nodes.
Proof. Suppose \( u_0 \in I \). Consider the tour beginning at \((u_0, s_0)\).

By checking the condition \((v, p) \notin \text{Visited} \) before calling \( \text{SCAN}(v, p) \) procedure (line 5 in the \( \text{SCAN} \) procedure), we guarantee that each pair \((u, q)\) is visited maximum once in the tour. Because there are finite pairs \((u, q)\), the tour will stop at finite steps.

Lemma 2.1 and Corollary 2.1 imply that:

By checking the condition in line 3 of the \( \text{SCAN} \), the result is extended with all pairs \((u_0, u)\) matching the query in which \( u \in O \).

Therefore, the algorithm returns the result of regular query \( R \) represented by query graph \( Q \) over data graph \( G \) with input set \( I \) and output set \( O \).

In each tour, each pair \((u, q)\) is visited maximum once so each data node is visited maximum \( k \) times. Each data edge is checked maximum \( k \) times (line 5. in the \( \text{SCAN} \) procedure). The complexity of each tour is \( O(m.k) \) and the complexity of the algorithm is \( O(|I|.m.k) \).

\(\square\)

2.3 Naive Joining Graph Algorithm

In the previous graph models, data graph is node-labeled graph while query graph is edge-labeled graph. The edge-labeled graph model for XML data sets is introduced in section 2.3.1. The state-data graph which is the result of joining an edge-labeled data graph with a query graph is studied in section 2.3.2. The result of regular query can be determined by the transitive closure of the state data graph. The naive joining graph algorithm is mentioned in section 2.3.3.

2.3.1 Edge-Labeled Data Graph

**Definition 2.6.** (Edge-labeled data graph) Let \( G = (V, E, \Sigma, r, \text{label}) \) be a data graph of XML data set \( \mathcal{D} \). The edge-labeled data graph of \( \mathcal{D} \) is a rooted directed edge-labeled graph \( G' = (V', E', \Sigma, r', \text{label}') \) in which:

i. \( r' \notin V \) is the root of the graph.

ii. \( V' = V \cup r', E' = E \cup (r', r) \).

iii. \( \text{label}' : E' \rightarrow \Sigma \) is edge-labeling function. For each \( e = (u, v) \in E' \) \((u \in V', v \in V)\), \( \text{label}'(e) \) is defined to be equal to \( \text{label}(v) \).

An example of node-labeled and edge-labeled graphs representing same XML data set is shown in Figure 2.3. The edge-labeled graph is created from the node-labeled graph by adding the new root (node 0) and an edge from the new root to the old root (edge (0,1)).
The label of an edge of the edge-labeled graph is the label of the destination node of that edge on the node-labeled graph.

The label of data node path, \( p' = u'_1u'_2 \ldots u'_k \), denoted by \( \text{label}(p') \in \Sigma^* \), is:

\[
\text{label}(p') := \text{label}(u'_0, u'_1).\text{label}(u'_1, u'_2) \ldots \text{label}(u'_{k-1}, u'_k).
\]

**Definition 2.7.** (Pair of data nodes matching the query) An ordered pair of data nodes \((u', v')\) matches regular expression \( R \) (denoted by \( u' \xrightarrow{R} \mathcal{G} \ xrightarrow{} v' \)) if there exists a data node path \( p' \) from \( u' \) to \( v' \) such that \( \text{label}(p') \in L(R) \).

**Definition 2.8.** (Result of regular query on edge-labeled data graph) The result of regular path expression \( R \) on the edge-labeled data graph \( \mathcal{G}' \) with input set \( I' \subseteq V' \), and output set \( O' \subseteq V' \) is:

\[
R^I_{O'}(\mathcal{G}', R) := \{(u', v') \in I' \times O'|u' \xrightarrow{R} \mathcal{G}' \ xrightarrow{} v'\}.
\]

### 2.3.2 State Data Graph

An edge-labeled graph \( \mathcal{G} \) is denoted by \( \mathcal{G} = (V, E, \Sigma, \text{label}) \) in which \( V \) is the set of nodes; \( E \subseteq V \times V \) is the set of edges; \( \Sigma \) is the set of label values; \( \text{label} : E \rightarrow \Sigma \) is a labeling function, each edge \( e \in E \) is labeled by \( \text{label}(e) \).

**Definition 2.9.** (Join operation of edge-labeled graphs) Let \( \mathcal{G}_1 = (V_1, E_1, \Sigma, \text{label}_1) \), \( \mathcal{G}_2 = (V_2, E_2, \Sigma, \text{label}_2) \) be edge-labeled graphs. \( \mathcal{G} = (V, E, \Sigma, \text{label}) \) edge-labeled graph is called the result of joining \( \mathcal{G}_1 \) and \( \mathcal{G}_2 \) if and only if:

i. \( V \subseteq V_1 \times V_2 \), and

ii. \( e = ((u_1, u_2), (v_1, v_2)) \in E \Leftrightarrow \exists e_1 \in E_1, e_2 \in E_2 : e_1 = (u_1, v_1) \land e_2 = (u_2, v_2) \land \text{label}_1(e_1) = \text{label}_2(e_2) = \text{label}(e) \).

**Definition 2.10.** (State data graph) The state data graph of an edge-labeled data graph and a query graph is the result of joining these graphs.
The state data graph of the edge-labeled and the query graph shown in Figure 2.3 and Figure 2.2 is depicted in Figure 2.4. Each edge of the state data graph can be considered as the result of "joining" a data edge and a query edge which have the same label. \(((u, s), (v, q))\) is an edge of the state data graph if and only if we can traverse the data graph and the query graph from \((u, s)\) to \((v, q)\) with the data edge \((u, v)\) and the query edge \((s, q)\). Generally, we can traverse the data graph and the query graph from \((u, s)\) to \((v, q)\) if and only if there exists a path over the state data graph from \((u, s)\) to \((v, q)\).

Suppose \(G = (V, E)\) is a graph. The transitive closure relation of graph \(G\) is:
\[
T(G) := \{(u, v) \in V \times V | \exists \text{ a path on } G \text{ from } u \text{ to } v\}
\]

**Proposition 2.2.** Let \(SD\) be the state-data graph of the query graph \(Q\) representing regular expression \(R\) and the edge-labelled data graph \(G', s_0, F\) be the start state and the set of final states of \(Q\). Then, we have:
\[
\mathcal{R}^I_Q(G', R) = \{(u', v') \in I' \times O' | \exists f \in F : ((u', s_0), (v', f)) \in T(SD)\}.
\]

**Proof.** \((u', v') \in \mathcal{R}^I_Q(G', R)\) if and only if \((u', v') \in I' \times O'\) and there exists a data node path \(p'\) from \(u'\) to \(v'\) matching the regular expression. \(label'(p')\) is accepted by the associated automata of query graph \(Q\) if and only if there exists a query node path from \(s_0\) to some \(f \in F\) final state of which the label is \(label'(p')\). Therefore, \(u' \xrightarrow{R}^G_{G'} v'\) if and only if there exists a data node path from \(u'\) to \(v'\) and a query node path from \(s_0\) to some final state \(f\).
which has the same label. In this case, there exists a path on the state data graph from \((u', s_0)\) to \((v', f)\). Finally,

\[
u' \xrightarrow{R_{G'}} v' \iff \exists f \in F : ((u', s_0), (v', f)) \in T(SD),
\]

and

\[
R^I_R(G', R) = \{(u', v') \in I' \times O' | \exists f \in F : ((u', s_0), (v', f)) \in T(SD)\}
\]

In the above example, if \(I' = \{s_0\}\) and \(O' = V'\)

\[
R^I_R(G', R) = \{(0, v') \mid ((0, s_0), (v', s_2)) \in T(SD)\} = \{(0, 4), (0, 5), (0, 6)\}.
\]

### 2.3.3 Naive Joining Graph Algorithm

**Definition 2.11.** (Relational representation of edge labeled graphs)

Let \(G = (V, E, \Sigma, \text{label})\) be an edge-labeled graph, \(D_V\) and \(D_\Sigma\) be the domains containing the values representing nodes and label value respectively. The relation \(R(\text{from} : D_V, \text{label} : D_\Sigma, \text{to} : D_V)\) is the relation representing \(G\) in which \((u, l, v)\) is a row of \(R\) if and only if \((u, v) \in E\) and \(\text{label}((u, v)) = l\).

An edge-labeled data graph \(G' = (V', E', \Sigma, r', \text{label}')\) is represented by two relations: \(D(d\text{from}, \text{label}, \text{dto})\) and \(DR(d\text{root})\). \(D\) represents the graph of the data graph; \(DR\) contains one row \((r)\) representing the root of the data graph. A query graph \(Q_R = (V_R, E_R, \Sigma, s_0, F)\) is represented by three relations: \(Q(q\text{from}, \text{label}, q\text{to})\), \(QR(q\text{root})\) and \(QF(q\text{final})\). \(Q\) represents the graph of the query graph; \(QR\) contains one row \((s_0)\) representing the root of the data graph; \(QF\) contains the set of rows representing the set of final states of the query graph.

**Proposition 2.3.** The state data graph of the query graph and the data graph can be presented by relation \(SD(d\text{from}, q\text{from}, \text{label}, \text{dto}, q\text{to})\) in which:

\[
SD = D \times_{\text{label}} Q
\]

and each record \((u, s, l, u', s')\) represents the edge of the state data graph from \((u, s)\) to \((u', s')\) of which label is \(l\).

The proposition is implied by the definition of the state data graph.
Figure 2.5: An example of the NAIVE-JOINING algorithm.

The NAIVE-JOINING-GRAPH algorithm computing the result with input set represented by relation $I(df from)$ and output set represented by relation $O(dto)$ is as follows:

1. Computing relation $SD(df from, qfrom, label, dto, qto)$ representing the state-data graph:

$$SD = D \Join_{\text{label}} Q.$$
2. Computing these pairs of state data nodes \((u, s_0, v, s)\) of the transitive closure of \(SD\) in which \(u\) is an element of the input set and \(s_0\) is the start state. These pairs are represented by relation \(T(d_{from}, q_{from}, dto, qto)\) and computed by following datalog rules:

\[
T(u, s, v, q) \leftarrow SD(u, s, l, r, t), I(u), QR(s), u = v, s = q
\]

\[
T(u, s, v, q) \leftarrow T(u, s, r, t), SD(r, t, l, v, q)
\]

3. Computing the result represented by relation \(R(d_{from}, dto)\). Each record \((u, v)\) of \(R\) matches the query, \(u \in I\) and \(v \in O\).

\[
R = \prod_{df_{from},dto} O \bowtie_{dto} T \bowtie_{T.qto=QF.qfinal} QF
\]

Let’s see the example shown in Figure 2.5. We represent and process the data graph shown in Figure 2.3 and the query graph shown in Figure 2.2 in relational database. The graph of the data graph is represented by relation \(D\). The query graph is represented by three relations \((Q, QR, QF)\). The input set \(I = \{r\}\) is represented by relation \(I\), and the output set \(O = V\) is represented by relation \(O\). Relation \(SD\) representing the state data graph equals to \(D \bowtie_{\text{label}} Q\). Relation \(T\) contains records \((0, s_0, u, s)\) of the transitive closure of \(SD\). Relation \(R\) represents the result.

**Proposition 2.4.** Relation \(R\) returned by the Naive-Joining-Graph algorithm is the result of query graph represented by the relations \((Q, QR, QF)\) over the data graph represented by the relation \(D\) with input set represented by the relation \(I\) and output set represented by the relation \(O\).

**Proof.** Proposition 2.3 implies that \(SD = D \bowtie_{\text{label}} Q\) represents the state-data graph. The transitive closure \(T'\) of \(SD\) can be computed by following datalog rules:

\[
T'(u, s, v, q) \leftarrow SD(u, s, l, r, t), u = v, s = q
\]

\[
T'(u, s, v, q) \leftarrow T'(u, s, r, t), SD(r, t, l, v, q)
\]

By adding two terms \(I(u)\) and \(QR(s)\) in the first rule, we have:

\[
T(u, s, v, q) \leftrightarrow T'(u, s, v, q) \land I(u) \land QR(s)
\]

Therefore, \(T\) contains and only contains the records \((u, s_0, v, q)\) such that:

\[
((u, s_0), (v, q)) \in T(SD) \land u \in I.
\]

Because of \(R = \prod_{df_{from},dto} O \bowtie_{dto} T \bowtie_{T.qto=QF.qfinal} QF\), \(R\) contains and only contains the records \((u, v)\) such that:

\[
\exists q \in F : T(u, s_0, v, q) \land v \in O.
\]
It is equivalent to:

\[ \exists q \in F : ((u, s_0), (v, q)) \in T(SD) \land (u, v) \in I \times O. \]

Proposition 2.2 implies that \( \mathcal{R} \) contains the result of the query graph over the data graph represented with the input set \( I \) and the output set \( O \). \( \square \)

The most expensive operation in the Naive-Joining-Graph algorithm is to execute the datalog rules. The join operations can be ignored as they are executed in linear time using hash join techniques. Similarly to the Naive-Graph-Traversal algorithm, the Naive-Joining-Graph algorithm has to find all ”reachable” pairs of state-data node from the input set and the the root of query graph. The complexity of these operations is \( O(|I| \cdot m \cdot k) \). With the Naive-Joining-Graph algorithm we can manage and process the XML data sets and regular queries by traditional database system.

2.4 Classification, Equivalence And Decomposition Of Regular Queries

2.4.1 Classification And Equivalence Of Regular Queries

Definition 2.12. (Empty-, primary-, finite- and universal queries) Let \( R \) be a regular expression over \( \Sigma \) alphabet.

i. \( R \) is empty if \( L(R) = L_{\epsilon} \).

ii. \( R \) is primary and denoted by \( R_\alpha \) if \( L(R) = \{ \alpha \} \) (\( \alpha \in \Sigma^* \) and \( \alpha \neq \epsilon \)).

iii. \( R \) is finite if \( |L(R)| < \infty \).

iv. \( R \) is universal denoted by \( R_\infty \) if \( L(R) = \Sigma^* \).

Let \( R, R' \) be regular expressions over the same alphabet. \( R \) is a sub-query of \( R' \) if \( L(R) \subseteq L(R') \).

Corollary 2.2. Let \( R \) be a sub-query of \( R' \). For arbitrary data graph \( \mathcal{G} \), input set \( I \) and output set \( O \). We have:

\[ \mathcal{R}_O^I(\mathcal{G}, R) \subseteq \mathcal{R}_O^I(\mathcal{G}, R'). \]

This corollary is implied by the definitions of the results of query over data graph.

Definition 2.13. (Equivalence of regular queries) We say that two regular path expression \( R, R' \) are equivalent on the data graph \( \mathcal{G} \) if:

\[ \forall I, O \subseteq V : \mathcal{R}_O^I(\mathcal{G}, R) = \mathcal{R}_O^I(\mathcal{G}, R'). \]
Let $R$ be a regular path expression. If there exists the value

$$l = \operatorname{Max}\{\text{length}(\alpha) | \alpha \in L(R)\},$$

$l$ is defined as the length of $R$. With the assumption that $\Sigma$ is finite, we have:

**Corollary 2.3.** There exists the length value of regular path expression $R$ if and only if $R$ is finite.

**Proof.** Certainly, if $R$ is finite, $L(R)$ is finite so there exists $l = \operatorname{Max}\{\text{length}(\alpha) | \alpha \in L(R)\}$. On the contrary, if there exists the value $l = \operatorname{Max}\{\text{length}(\alpha) | \alpha \in L(R)\}$, we have:

$$L(R) \subseteq \Sigma^l.$$ $\Sigma^l$ is finite since $\Sigma$ is finite. Therefore, $R$ is finite. □

Let $L(R, i) = \{\alpha \in L(R) | \text{length}(\alpha) \leq i\}$ ($i \in \mathbb{N}$). Clearly, $L(R, i)$ is finite, and $L(R, i) \subseteq L(R)$. Because $L(R, i)$ is finite, $L(R, i)$ is also regular language and can be represented by a finite regular path expression. The finite regular path expression representing $L(R, i)$ is called the not longer than $i$ sub-query of $R$. It is denoted by $R(i)$.

**Proposition 2.5.** Let $\mathcal{G}$ be a data graph. For each $R$ regular expression, there exists $h \in \mathbb{N}$ such that $R(i)$ and $R$ are equivalent on $\mathcal{G}$. Concretely,

i. If $\mathcal{G}$ is an acyclic graph with the diameter $d$, $h$ is chosen so that $h = d$.

ii. If $\mathcal{G}$ is an arbitrary graph, $h$ is chosen so that $h = nk$ in which $n$ is the number of data nodes and $k$ is the number of query nodes of the query graph of $R$.

**Proof.** Let $I, O \subseteq V$ arbitrary input, output set. Because $R(h)$ is a sub-query of $R$, we have:

$$\mathcal{R}_O^I(\mathcal{G}, R(h)) \subseteq \mathcal{R}_O^I(\mathcal{G}, R).$$

Therefore, the proposition is true if we can show that:

$$\exists h \in \mathbb{N} : \mathcal{R}_O^I(\mathcal{G}, R) \subseteq \mathcal{R}_O^I(\mathcal{G}, R(h)).$$

Consider $h$ is chosen in the proposition. Let $(u, v) \in \mathcal{R}_O^I(\mathcal{G}, R)$. We will show that $(u, v) \in \mathcal{R}_O^I(\mathcal{G}, R(h))$.

In the case that $\mathcal{G}$ is acyclic and $h$ is chosen so that $h = d$. Suppose $p$ is the data node path from $u$ to $v$ in which $\alpha = \text{label}(p) \in L(R)$. Because $\mathcal{G}$ is acyclic, the length of an arbitrary path is not greater than the graph diameter. Therefore, $\text{length}(\alpha) = \text{length}(p) \leq d = h$. It implies $\alpha \in L(R, h)$. Hence,

$$(u, v) \in \mathcal{R}_O^I(\mathcal{G}, R(h)).$$

In the general case, $h$ is chosen so that $h = nk$. Since $(u, v) \in \mathcal{R}_O^I(\mathcal{G}, R)$, Corollary 2.1 implies that there exists final state $f$ such that $(u, v)$ matches $(s_0, f)$. Suppose $l$ is the
distance of \((u, v)\) and \((s_0, f)\). Suppose \(p = u_0u_1\ldots u_{l-1}\) is the data node path from \(u\) to \(v\) \((u_0 = u, u_{l-1} = v)\), and \(p_q = q_0q_1\ldots q_l\) is the query node path from \(s\) to \(f\) \((q_0 = s_0, q_l = f)\) such that \(p\) matches \(p_q\). Then, we have:

\[
\forall i \in \{0, 1, \ldots, l - 1\} : \ (u_i, q_i) \in Visited(u_0),
\]

where \(Visited(u_0)\) is defined in subsection 2.2.

\((p, p_q)\) is the shortest matching pair of data node path and query node path of \((u, v)\) and \((s_0, f)\). Hence, \(\{(u_i, q_i)\}\) are different from the others. Therefore,

\[
label(p) = l = |\{(u_i, q_i)\}| \leq |Visited(u_0)| \leq nk = h.
\]

Hence, \(label(p) \in L(R, h)\). It implies that

\[(u, v) \in R^I_O(G, R(h)).\]

\[\square\]

### 2.4.2 Decomposition Of Regular Queries

**Proposition 2.6.** Let \(R, R'\) be regular expression over data graph \(G\). Then, we have:

\[
R^I_O(G, R\mid R') = R^I_O(G, R) \cup R^I_O(G, R').
\]

**Proof.** There exists a data node path \(p\) from \(u\) to \(v\) matching \(R\mid R'\) if and only if there exists a data node path from \(u\) to \(v\) matching \(R\) or a data node path from \(u\) to \(v\) matching \(R'\). Therefore,

\[
R^I_O(G, R\mid R') = R^I_O(G, R) \cup R^I_O(G, R').
\]

\[\square\]

A regular queries can be decomposed into primary queries.

**Proposition 2.7.** Let \(G\) be a data graph. For each regular expression \(R\), there exists a finite set \(L \subseteq \Sigma^*\) such that:

\[
R^I_O(G, R) = \bigcup_{\alpha \in L} R^I_O(G, R_\alpha).
\]

**Proof.** Let \(L\) be the language denoted by \(R(h)\) which is equivalent to \(R\) over \(G\) (Proposition 2.5). Clearly, \(L\) is finite. Therefore,

\[
R^I_O(G, R) = R^I_O(G, R(h)) = \bigcup_{\alpha \in L} R^I_O(G, R_\alpha).
\]

\[\square\]
Let’s see the example of decomposing the regular expression $R = a.(b|c)^*.a$ shown in Figure 2.2 over the data graph $G$ shown in Figure 2.3. $h = 4$ as $G$ is acyclic and its diameter is 4. $\mathcal{L} = \{abba, abca, acca, acba, aba, aca, aa\}$. $R$ is equivalent to

$$R_{abba} \big| R_{acba} \big| R_{acca} \big| R_{aca} \big| R_{aba} \big| R_{aa}$$

over $G$ and

$$\mathcal{R}_O^I(G, R) = \mathcal{R}_O^I(G, R_{abba}) \cup \mathcal{R}_O^I(G, R_{abca}) \cup \mathcal{R}_O^I(G, R_{acba}) \cup \mathcal{R}_O^I(G, R_{acca}) \cup \mathcal{R}_O^I(G, R_{aca}) \cup \mathcal{R}_O^I(G, R_{aba}) \cup \mathcal{R}_O^I(G, R_{aa}).$$

2.5 Summary

In this chapter, we have studied on the preliminary of processing regular queries over the XML data sets. Both XML data sets and regular queries can be represented by the directed labeled graphs: data graph and query graph. Data graph is a rooted graph whose basic structure is a tree. Data nodes represent XML nodes, and data edges represent the relationships between XML nodes. Data nodes (in the case of node-labeled data graph) or data edges (in the case of edge-labeled data graph) are labeled to describe the semantics of the data. Query graph is the transition graph of the automata computing the regular query.

A pair of data nodes matches the regular query if there exists a data node path from the first node to the second node matching the query. Based on this definition, we define the natural result and the result with a given input set and output set of a regular query over a data graph. The results of a regular query over a data graph can be computed by the Naive-Graph-Traversal algorithm using node-labeled data graph or the Naive-Joining-Graph algorithm using edge-labeled data graph. In the case of the Naive-Graph-Traversal algorithm, both of data graph and query graph are traversed together to find all matching pairs of query node path and data node path. In the case of the Naive-Joining-Graph algorithm, query graph, data graph, input set and output set are represented by relations. The result is determined by computing the transitive closure of the state-data graph which is the result of joining the data graph and the query graph. These two algorithms have the same complexity, but the Naive-Joining-Graph
algorithm can store, manage and process data graphs and query graphs in traditional database system.

Finally, for each regular query there exists a finite sub-query which is equivalent with that query on a given data graph. As a result, with a given data graph, each query can be decomposed into the primary regular queries.
Chapter 3

Structural Indexes, DL-Indexes

The structural indexes [6, 11–13, 18, 20] have been proposed to summarize the structure of XML data and speed up path expression evaluation. A structural index is usually much smaller than the original data graph so that the complexity of the query evaluation over the index graph is usually much lower than that over the data graph. However, the index graph is only equivalent to the data graph over a given set of queries called query load. For each query in the query load, the result of the query over the index graph coincides with that over the data graph. In the case that the processed query is not supported by the index graph, the results over the index graph may be different from that over the data graph. The query is processed over the index graph first, and then the result is validated on the data graph.

Based on the query load, structural indexes are classified into two classes: static structural indexes [12, 18, 26] and dynamic structural indexes [6, 11]. The query loads of the stable structural indexes are unchangeable. Contrarily, the query loads of dynamic structural indexes are adaptive and fitted with frequently used queries. The dynamic structural indexes overcome the static ones according to the average processing cost and the sizes of the indexes [6, 11]. This is because the dynamic structural indexes are adaptive. The most popular structural indexes are the 1-index [18], the $A(k)$-index [12] (static structural indexes), the $D(k)$-index [6], the $M(k)$-index and the $M^*(k)$-index [11] (dynamic structural indexes).

Data nodes in the same index node of most structural indexes must have the same label. These structural indexes are called static labeling structural indexes since they treat the label values uniformly. In the case that the query load is considered, if the queries
especially contain the wild card or alternation card, the significance of labels are different. For an example, suppose there are some label values which are not used in any query of the query load. These unused label values should be represented by only one special label noting the uselessness. The structural index will be more efficient if the dynamic labeling approach (DL for short) is used.

In this chapter, a mathematical background based on the foundation of partitions and the split operation for structural indexes is studied. From this mathematical background, every classical structural index can be improved to support dynamic labeling. The DL-1-index developed from the 1-index and the DL-A* (k)-index developed from the A(k)-index are investigated. DL-1-index and the DL-A* (k)-index illustrate the techniques and ideas, which are used for improving a static labeling structural index to support dynamic labeling.

The near primary queries (given by formula \( {^*\alpha} \)) hold an important role in the query decomposition and processing. The 1-indexes base on the notion of bisimulation, in which nodes in the same index node have the same set of incoming label paths. The query load is the set of near primary queries. The \( A(k) \)-indexes base on the notion of \( k \)-bisimilarity, which can be considered as a weakening of the bisimulation. All nodes in the same index node have the same set of incoming label paths which are not longer than \( k \). Thus, with the \( A(k) \)-index, the query load is the set of near primary queries which are not longer than \( k \). In the case that the near primary query is longer than \( k \), we have to validate the answer on the data graph. The equivalent relation in the \( A(k) \)-index is weaker than that in the 1-index. Hence, the size of the \( A(k) \)-index is smaller than that of the 1-index. Since the length of queries is often short in practice, if \( k \) is chosen suitably the queries evaluation over the \( A(k) \)-index is more efficient than that over the 1-index \([12]\).

The simple alternation path expressions (or SAPE for short) are general types of the near primary queries. A SAPE may contain the wild or alternation cards. The query load of the DL-1-index and the DL-A* (k)-index is a finite set of SAPE queries. The construction and refinement of the DL-indexes base on our new results of studying split operation over partitions. Dynamic labeling permits that index node is labeled more than one value. The dynamic labeling not only makes the size of the index graph smaller but also reduces the evaluation cost over the query load.
The chapter is organized as follows. Section 3.1 is the preliminary. The basic definitions of structural indexes and SAPE queries are introduced. Section 3.2 investigates the basic mathematical of structural indexes. Partitions and the split operation are studied. Section 3.3 introduces and studies the 1-index and the DL-1-index. Section 3.4 investigates the A(k)-index and the DL-A*(k)-index. Finally, Section 3.5 concludes the chapter.

3.1 Preliminaries

3.1.1 Near Primary Queries And Simple Alternation Path Expressions

Definition 3.1. (Near primary query) Regular expression $R$ is a near primary query if

$$R = R_\infty R_\alpha$$

where $R_\infty$ is universal query, and $R_\alpha (\alpha \in \Sigma^*)$ is a primary query. The length of $R$ is defined as $\text{length}(\alpha) - 1$.

Suppose $S \subseteq \Sigma$ is a set of label values, and $S \neq \emptyset$. Then, we define:

i. $R_S := _ \_ \_ \_$ if $S = \Sigma$,

ii. $R_S := (a_1|a_2|\ldots|a_m)$ if $S \neq \Sigma$ and $S = \{a_1, a_2, \ldots, a_m\}$.

Definition 3.2. (SAPE) A simple alternation path expression (SAPE for sort) $R$ is given by a sequence $S_0 S_1 \ldots S_l$ if

$$R = R_\infty R_{S_0} R_{S_1} \ldots R_{S_l}$$

in which $l \in \mathbb{N}$, $S_i \subseteq \Sigma$, $S_i \neq \emptyset$ ($i = 0, 1, \ldots, l$). The length of $R$ is defined as $l$.

Certainly, a near primary query is a special case of SAPE in which $|S_i| = 1$ for each $i = 0, \ldots, l$. The query graph of the SAPE $S_0 S_1 \ldots S_l$ is shown in Figure 3.1. The query graph has $(l + 2)$ query nodes: $s_0, s_1, \ldots, s_{l+1}$. $s_0$ is the start state and $s_{l+1}$ is a final state. For each $a_i \in S_i$ ($i = 0, \ldots, l$), there is a query edge from $s_i$ to $s_{i+1}$ labeled by $a_i$. Moreover, for each $a \in \Sigma$, there is an edge from $s_0$ to $s_0$ labeled by $a$.

We say that data node path $p = u_0 u_1 \ldots u_l$ matching the end of the SAPE given by $S_0 S_1 \ldots S_l$ if $\text{label}(u_i) \in S_i$ ($i = 0, 1, \ldots, l$). Let $P_R$ be the set of data node path matches the end of the SAPE given by $S_0 S_1 \ldots S_l$. $T_{p_i}(R)$ denotes the set of the $i$-th data nodes $u_i$ of $p = u_0 u_1 \ldots u_l \in P_R$. $T_{p_i}(R)$ is called the target set of $R$ and it is also denoted by $T_{p_i}(R)$. 
Proposition 3.1. Let \( R \) be a SAPE given by \( S_0 S_1 \ldots S_l \). Then, we have:
\[
R(\mathcal{G}, R) = T_\mathcal{G}(R).
\]

Proof. \( u \in R(\mathcal{G}, R) \) if and only if \((r, f)\) matches \((s_0, s_l)\). This is equivalent to that there exists a data node path \( v_0 v_1 \ldots v_t u_0 u_1 \ldots u_l \) such that:
\[
v_0 = r \land u_l = u,
\]
and
\[
s_0 \in E_R(s_0, \text{label}(u_i)) \quad (i = 0, 1, \ldots, t),
\]
and
\[
s_{i+1} \in E_R(s_i, \text{label}(u_i)) \quad (i = 0, 1, \ldots, l).
\]
According to the query graph of \( R \), we have:
\[
\forall a \in \Sigma : s_0 \in E_R(s_0, a)
\]
and
\[
s_{i+1} \in E_R(s_i, a) \iff a \in S_i \quad (i = 0, 1, \ldots, l)
\]
Hence, \( u \in R(\mathcal{G}, R) \) if and only if there exists a data node path \( u_0 u_1 \ldots u_l \) such that \( u_l = u \) and \( \text{label}(u_i) \in S_i \) for each \( i = 0, 1, \ldots, l \). It implies that
\[
R(\mathcal{G}, R) = T_\mathcal{G}(R)
\]
\[\square\]

3.1.2 Structural Indexes

Definition 3.3. (Structural index) Let \( \mathcal{G} = (V, E, \Sigma, r, \text{label}) \) be a data graph. A structural index of the data graph \( \mathcal{G} \) is a labeled directed graph
\[
I(\mathcal{G}) = (V_I, E_I, \Sigma)
\]
in which:

i. \( V_I \) is the set of index nodes. Each index node \( U \) is a non empty set of data nodes \( (U \subseteq V) \) and is labeled by \( \text{label}(U) = \bigcup_{u \in U} \text{label}(u) \). Moreover, \( V = \bigcup_{U \in V_I} U \).

ii. \( E_I \subseteq V_I \times V_I \) is the set of index edges. For each index edge \((U, U')\) there exists a data edge \((u, u')\) in which \( u \in U \) and \( u' \in U' \).
Data node $u$ is simulated by the index node $U$ if $u \in U$. Data edge $(u, u')$ is simulated by the index edge $(U, U')$ if $u \in U$ and $u' \in U'$. Data node path $u_0u_1\ldots u_l$ ($l \geq 1$) is simulated by the index node path $U_0U_1\ldots U_l$, if for each $i = 0, \ldots, l - 1$ $(u_i, u_{i+1})$ is simulated by $(U_i, U_{i+1})$.

For each $W \subseteq V_I$, we define:

$$\text{Extent}(W) := \bigcup_{U \in W} U.$$ 

In the definition of structural indexes, the condition $V = \text{Extent}(V_I)$ guarantees that every data node is simulated by some index node. However, maybe a data edge (data node path) is not simulated by any index edge (index node path).

**Proposition 3.2.** Suppose index nodes are disjointed, and each data edge is simulated by some index edge. Then, each data node, data edge and data node path is simulated by one and only one index node, index edge and index node path, respectively.

**Proof.** Because index nodes are disjointed, a data node can not be simulated by two different index nodes (otherwise, the data node is the common element of these index nodes). Therefore, each data node is simulated by one and only one index node.

We assume data edge $(u, u')$ is simulated by the index edges $(U, U')$ and $(U_1, U'_1)$. Because $u \in U \cap U_1$, $u' \in U' \cap U'_1$ and each data node is simulated by one and only one index node, $U = U_1$ and $U' = U'_1$. It implies that each data edge is simulated one and only one index edge.

Each data node path $p = u_0u_1\ldots u_l$ is simulated by one and only one index node path $U_0U_1\ldots U_l$, in which $u_i$ is simulated by $U_i$ ($i = 0, 1, \ldots, l$).

**Definition 3.4.** (Matching of query node path and index node path) Suppose $p_I = U_1\ldots U_l$ is a index node path, and $p_q = q_0\ldots q_l$ is a query node path. We say that $p_I$ matches $p_q$ if

$$\forall i = 1, \ldots, l : q_i \in E_R(q_{i-1}, \text{label}(U_i)).$$

In this case, we also say that $(U_1, U_l)$ matches $(q_0, q_l)$.

An index node path of the index graph $I_G$, $U_0U_1\ldots U_l$, matches the end of SAPE given by $S_0S_1\ldots S_l$ if $\text{label}(U_i) \cap S_i \neq \emptyset$ for each $0 \leq i \leq l$.

**Definition 3.5.** (Target set and result of a SAPE over structural index) The target set of $R$ on $I_G$ denoted by $T_{I_G}(R)$ is the set of the end nodes of the index node paths in $I_G$ which match the end of $R$. The result of $R$ over $I_G$ denoted by $T_{I_G,G}(R)$ is $\text{Extent}(T_{I_G}(R))$. 
Definition 3.6. (Safeness and soundness of structural indexes) An index graph $I_G$ is safe with a SAPE $R$ if for each $u \in T_G(R)$ and $u$ is simulated by $U$, we have:

$$U \in T_{I_G}(R).$$

An index graph $I_G$ is sound with a SAPE $R$ if for each $U \in T_{I_G}(R)$ and $u$ is simulated by $U$, we have:

$$u \in T_G(R).$$

Corollary 3.1. If $I_G$ is safe and sound with a SAPE $R$, we have:

$$T_{I_G,G}(R) = T_G(R).$$

Proof. Clearly, if $I_G$ is safe with $R$, $T_{I_G}(R) \subseteq T_{I_G,G}(R)$. On the contrary, if $I_G$ is sound with $R$, $T_{I_G,G}(R) \subseteq T_G(R)$. Hence, if $I_G$ is safe and sound with $R$, we have:

$$T_{I_G,G}(R) = T_G(R).$$

Definition 3.7. (Equivalence of index graph and data graph) In the case that the results of query $R$ over the index graph and those over the data graph are coincide, we say that the index graph is equivalent to the data graph over $R$.

Definition 3.8. (Soundness of target index node) A target index node $U$ of $R$ is sound if there exists an index node path $U_0U_1\ldots U_l$ such that $U_i = U$ and $\text{label}(U_i) \subseteq S_i$ for each $i = 0, \ldots, l$.

Proposition 3.3. Let $R$ be a SAPE given by $S_0S_1\ldots S_l$, $I_G$ be a structural index of data graph $G$.

1. $I_G$ is safe with $R$ if each data node path $U_0U_1\ldots U_l$ is simulated by some index node path $u_0u_1\ldots u_l$.
2. $I_G$ is sound with $R$ if two following conditions are satisfied:
   i. For each index edge $(U, U')$: $U' \subseteq \text{Succ}(U)$.
   ii. All target index nodes $U$ of $R$ on $I_G$ are sound.

Proof. Suppose that $I_G$ satisfies the condition in (1).

Let $u \in T_G(R)$ and $u_0u_1\ldots u_l$ be the data node path matching the end of $R$ in which $u_l = u$. Suppose that $u_0u_1\ldots u_l$ is simulated by some index node path $U_0U_1\ldots U_l$.

Label($u_i$) $\in S_i$ ($U_0u_1\ldots u_l$ matches the end of $R$) and label($u_i$) $\in$ label($U_i$) ($u_i \in U_i$) so label($U_i$) $\cap S_i \neq \emptyset$ ($i = 0, 1, \ldots, l$). Thus $U_0U_1\ldots U_l$ also matches the end of $R$ and $U_l \in T_{I_G}(R)$. Hence, $I_G$ is safe with $R$.

Suppose that $I_G$ satisfies two conditions in (2).

Let $U \in T_{I_G}(R)$ and $U_0U_1\ldots U_l$ be the index node path matching the end of $R$ in which $U_l = U$ and $\text{label}(U_i) \subseteq S_i$ for each $i = 0, 1, \ldots, l$. Since $(U_i, U_{i+1})$ is an index edge, we have:

$$U_{i+1} \subseteq \text{Succ}(U_i) \quad (i = 0, 1, \ldots, l - 1).$$
It implies that for each \( u_{i+1} \in U_{i+1} \) there exists \( u_i \in U_i \) such that \((u_i, u_{i+1})\) is a data edge. Therefore, for each \( u_i \in U_i \) there exists a data node path \( u_0 \ldots u_i \) such that \( u_i \in U_i \) \((i = 0, 1, \ldots, l)\). \( \text{label}(U_i) \subseteq S_i \) implies \( \text{label}(u_i) \in S_i \). Thus, \( u_0 u_1 \ldots u_l \) also matches the end of \( R \) and \( u_l \in T_G(R) \). Hence, \( I_G \) is sound with \( R \).

**Proposition 3.1.3** provides us the sufficient conditions for the soundness and the safeness of structural indexes. The condition in (1) is called the sufficient safeness condition. Two conditions in (2) are called the first and second sufficient soundness condition.

### 3.2 Partitions And Split Operation

#### 3.2.1 Partitions

**Definition 3.9.** (Partition) \( B = \{B_i | i \in I, B_i \subseteq V, B_i \neq \emptyset \} \) is a partition over \( V \) if

i. \( V = \bigcup_{i \in I} B_i \), and

ii. \( \forall i, j \in I, i \neq j : B_i \cap B_j = \emptyset \).

**Definition 3.10.** (Refinement relation) Let \( C \) be a set of subsets of \( V \), \( B \) be a partition over \( V \). \( B \) is a refinement of \( C \) if

\[
\forall C \in C, B \in B : B \subseteq C \vee B \cap C = \emptyset.
\]

Clearly, \( B^* = \bigcup_{u \in V} \{u\} \) is a refinement of any set of subsets of \( V \).

**Definition 3.11.** (Refinement relation over partitions) Let \( B \) and \( B' \) be partitions. \( B' \succeq B \) if \( B' \) is a refinement of \( B \).

**Remark 3.1.** \( \succeq \) holds following properties:

1. \( \succeq \) is reflexive, transitive and antisymmetric relation.

2. \( B' \succeq B \iff \forall B \in B, B \text{ is the union of some elements of } B' \).

**Definition 3.12.** (The coarsest refinement) Let \( C \) be a set of subsets of \( V \). \( B \) is called the coarsest refinement of \( C \) if:

i. \( B \) is a refinement of \( C \), and

ii. If partition \( B' \) is a refinement of \( C \), it holds \( B' \succeq B \).

\( \text{CRF}(C) \) denotes the coarsest refinement of \( C \).

The following proposition shows the existence and the recursive construction of the coarsest refinement.

**Proposition 3.4.** Let \( B \) be a partition, \( C \) be a subset of \( V \), \( C \) and \( C' \) be set of subsets of \( V \). Moreover, we assume that there exists \( \text{CRF}(C') \). Then, we have:

1. \( \text{CRF}(\{C\}) = \{C, V \setminus C\} \setminus \{\emptyset\} \).

2. \( \text{CRF}(B \cup \{C\}) = \bigcup_{B \in B} \{B \setminus C, B \cap C\} \setminus \{\emptyset\} \).

3. \( \text{CRF}(C' \cup C) = \text{CRF}(\text{CRF}(C') \cup C) \).
**Proof.** (1) and (2) are implied directly from the definitions of the coarsest refinement and the partitions imply.

To prove (3), we show that for any partition $B$, $B$ is a refinement of $C' \cup C$ if and only if $B$ is a refinement of $\text{CRF}(C') \cup C$.

Let $B$ be a refinement of $\text{CRF}(C') \cup C$, and $P$ be an element of $B$. We will show that:

- $\forall C' \in C' : P \subseteq C'$ or $P \cap C' = \emptyset$ (a)
- $\forall C \in C : P \subseteq C$ or $P \cap C = \emptyset$ (b)

Because of $B \succeq \text{CRF}(C')$, there exists $B' \in \text{CRF}(C') : P \subseteq B'$. Since $\forall C' \in C' : B' \subseteq C'$ or $B' \cap C' = \emptyset$ and $P \subseteq B'$, (a) is true.

Since $B$ is a refinement of $\text{CRF}(C') \cup C$, $B$ is a refinement of $C$. Hence, (b) is true.

(a) and (b) imply that $B$ is a refinement of $C' \cup C$.

Hence, (3) is true.

Trivially, $\text{CRF}(\{\emptyset\}) = \text{CRF}(\{V\}) = \{V\}$. Suppose $V = \{1, 2, 3, 4, 5, 6\}$. Then, we have

$$\text{CRF}([1, 2], [2, 3], [3, 4, 5]) = \{\{1\}, \{2\}, \{3\}, \{4, 5\}, \{6\}\}.$$

**Remark 3.2.** The CRF function holds following properties:
1. $\text{CRF}(C) = C \iff C$ is a partition over $V$.
2. If $C \subseteq C'$, $\text{CRF}(C') \succeq \text{CRF}(C)$.
3. $\text{CRF}(C' \cup C) = \text{CRF}(\text{CRF}(C') \cup \text{CRF}(C))$.
4. If $\text{CRF}(C') \succeq \text{CRF}(C)$, we have:

$$\text{CRF}(C' \cup C) \succeq \text{CRF}(C' \cup C).$$

### 3.2.2 Split Operation

Let $C$ be a subset of $V$ and $C$ be a set of subsets of $V$. Then, we define:

$$\text{Succ}^0(C) := C, \quad \text{Succ}^{k+1}(C) := \text{Succ}^{k}(\text{Succ}^{k}(C)),$$

and

$$\text{Succ}^{k}(C) := \{C' \exists C \in C : C' = \text{Succ}^{k}(C)\} \quad (k \in \mathbb{N}).$$
Definition 3.13. (split, split\(_i\) and split\(_\infty\) operations) Suppose \(C\) be a subset of \(V\). The split operation is defined as follows:

\[
\text{split}(C) := CRF(C \cup \text{Succ}(CRF(C))).
\]

The split\(_i\) and split\(_\infty\) operations are defined inductively:

\[
\text{split}_0(C) := CRF(C), \quad \text{split}_{k+1}(C) := \text{split}(\text{split}_k(C)) \quad (k \in \mathbb{N}).
\]

Suppose there exists \(k_0 \in \mathbb{N}\) such that:

\[
\forall k \in \mathbb{N}, k \geq k_0 : \text{split}_k(C) = \text{split}_{k_0}(C).
\]

Then, we define:

\[
\text{split}_\infty(C) := \text{split}_{k_0}(C).
\]

The existence of \(k_0\) in the definition of split\(_\infty\) follows by the fact that split\(_{k+1}\)(\(C\)) is a refinement of split\(_k\)(\(C\)) and \(V\) is finite.

We say that partition \(B\) is stable if split\((B) = B\).

Proposition 3.5. Let \(B, B'\) be partitions and \(B' \succeq B\). Then, we have:

1. If \(B' \succeq \text{split}(B)\), we have:

\[
\forall B' \in B', B \in B : B' \cap \text{Succ}(B) = \emptyset \vee B' \subseteq \text{Succ}(B).
\]

2. \(\text{split}_k(B') \succeq \text{split}_k(B) \quad (k \in \mathbb{N}).\)

3. \(\text{split}_\infty(B') \succeq \text{split}_\infty(B).\)

4. If \(B'\) is stable, we have:

\[
B' \succeq \text{split}_\infty(B) \succeq B.
\]

Proof. Because \(B' \succeq \text{split}(B) \succeq CRF(\text{Succ}(B))\) (Remark 3.2), (1) is true.

(2) is true when \(k = 0\) as split\(_0(B') = B' \succeq B = \text{split}_0(B)\). Because of \(B' \succeq B\), each \(B \in B\) is the union of some elements of \(B'\) and \(\text{Succ}(B)\) is the union of some elements of \(\text{Succ}(B')\). Therefore, \(CRF(\text{Succ}(B')) \succeq CRF(\text{Succ}(B))\) and \(\text{split}(B') \succeq \text{split}(B)\) (Remark 3.2).

Using induction we have \(\text{split}_k(B') \succeq \text{split}_k(B)\) for all \(k \in \mathbb{N}_0\).

(2) implies (3) and (3) implies (4) directly.

Let \(m\) be the number of edges and \(n\) be the number of nodes of the data graph. The split\((C)\) can be computed with the complexity \(O(m + n)\) by scanning only one time all edges and nodes. The split\(_\infty\)(\(C\)) can be computed by using PT algorithm [19] with the complexity \(O(m \log n)\).
3.3 1-Index And DL-1-Index

3.3.1 1-Index And Stable Structural Indexes

The foundation of the 1-index [18] is the bisimilarity relation defined as follows:

**Definition 3.14.** (Bisimilarity) Two data nodes u and v are strong bisimilar \( (u \approx v) \) if:

i. they have the same label, and

ii. if \( (u', u) \) is an edge, then there exists \( v' \) such that \( u' \approx v' \) and \( (v', v) \) is also an edge, and vice versa.

Let \( P_c \) be the partition determined by the equivalence \( \approx_c \) in which \( u \approx_c v \iff \text{label}(u) = \text{label}(v) \). Suppose \( P \) is a partition determined by a bisimilarity relation. In the definition of the bisimilarity, the first condition implies that \( P \succeq P_c \). The second condition implies that \( P \) is stable. Proposition 3.5 implies that \( P \succeq \text{split}_\infty(P_c) \). Generally,

**Remark 3.3.** \( P \) is a stable refinement of \( \text{split}_\infty(P_c) \) if and only if the associated equivalence relation is bisimilarity.

The set of index nodes of the 1-index is \( \text{split}_\infty(P_c) \). The 1-index is a special case of the stable structural indexes which are defined as follows:

**Definition 3.15.** (Stable structural index) \( I_G = (V_I, E_I, \Sigma) \) is a stable structural index created by the partition \( V_I \) if:

i. \( V_I \) is a stable partition, and

ii. there is an index edge \( (U, U') \) if there exists a data edge \( (u, u') \) such that \( u' \in U' \) and \( u \in U \).

**Proposition 3.6.** All stable structural indexes satisfy the sufficient safeness condition and the first sufficient soundness condition for any SAPE.

**Proof.** As \( V_I \) is a partition and all edges are simulated, Proposition 3.2 implies that every data node path is simulated by one and only one index node path. Hence, all stable structural indexes satisfy the sufficient safeness condition for any SAPE.

Because \( V_I \) is a stable partition, Proposition 3.5 implies that

\[
\forall U, U' \in V_I : \quad U' \cap \text{Succ}(U) = \emptyset \lor U' \subseteq \text{Succ}(U).
\]

Suppose \( (U, U') \) is an index edge. According to the definition of structural indexes, we have: \( U' \cap \text{Succ}(U) \neq \emptyset \). Hence, \( U' \subseteq \text{Succ}(U) \). Therefore, all stable structural indexes satisfy the first sufficient soundness condition for any SAPE.

A stable structural index \( I_G \) supports a SAPE \( R \), if it satisfies the second sufficient soundness condition. Clearly, if \( I_G \) supports \( R \) then it is safe and sound with \( R \).
**Proposition 3.7.** The 1-index is safe and sound with any SAPE.

*Proof.* For each index node $U_i$, $|\text{label}(U_i)| = 1$. Hence, if $\text{label}(U_i) \cap S_i \neq \emptyset$, $\text{label}(U_i) \subseteq S_i$. Therefore, the 1-index satisfies the second sufficient soundness condition for any SAPE. Hence, the 1-index is safe and sound with any SAPE. □

**Definition 3.16.** (Refinement of a stable structural index) A stable structural index graph $I_G$ is a refinement of the stable structural index graph $I_G'$, if the partition created by the index nodes of $I_G$ is a refinement of that of $I_G'$.

Let $\mathcal{P}_a = \{V\}$ be the coarsest partition over $V$. The coarsest stable structural index is the stable structural index created by $\text{split}_\infty(\mathcal{P}_u)$. Clearly, every stable structural index is a refinement of the coarsest stable structural index.

We study on the inheritance of the support of a SAPE between a stable structural index and its stable refinements in the following proposition.

**Proposition 3.8.** Let $I_G$, $I_G'$ be stable structural index graphs, in which $I_G$ is a refinement of $I_G'$, and $R$ be a SAPE given by $S_0 \ldots S_t$. We have:

1. Suppose $U_0 \ldots U_l$ is an index node path of $I_G$ matching the end of $R$, and $U'_0, \ldots, U'_l$ are the index nodes of $I_G'$ such that $U_i \subseteq U'_i$, $0 \leq i \leq l$. Then, $U'_0 \ldots U'_l$ is also an index node path of $I_G'$ matching the end of $R$.

2. If $I_G'$ supports $R$, $I_G$ supports $R$, too.

*Proof.* 1. From the definition of structural indexes, if $U_i U_{i+1}$ is an index edge of $I_G$, $U'_i U'_{i+1}$ is also an index edge of $I_G'$. It implies that $U'_0 \ldots U'_l$ is an index node path of $I_G'$. Because of $\text{label}(U_i) \cap S_i \neq \emptyset$ and $\text{label}(U_i) \subseteq \text{label}(U'_i)$, $\text{label}(U'_i) \cap S_i \neq \emptyset$ and $U'_0 \ldots U'_l$ matches the end of $R$.

2. Let $U$ be a target index node of $R$ on $I_G$, and $U'$ be an index node of $I_G'$ such that $U \subseteq U'$. (1) implies that $U'$ is also a target index node of $R$ on $I_G'$. Because $I_G'$ supports $R$, there exists an index node path $U'_0 \ldots U'_l$ matching $R$, in which $\text{Label}(U'_i) \subseteq S_i$, $0 \leq i \leq l$ and $U'_i = U'$. Let $u \in U$ be a data node, and a $u_0 \ldots u_t$ be the node path, such that $u_t \in U'_i$ and $u_t = u$. Let $U_0 U_1 \ldots U_l$ be the index node path of $I_G$ simulating $u_0 u_1 \ldots u_t$. Since $u$ is simulated by $U$ and $U_i$, $U = U_i$. $u_i \in U_i \cap U'_i$ implies $U_i \cap U'_i \neq \emptyset$. Hence, $U_i \subseteq U'_i$ and $\text{label}(U_i) \subseteq \text{label}(U'_i) \subseteq S_i$. $U$ is a sound target index node. Thus, $I_G$ supports $R$. □

### 3.3.2 DL-1-index

DL-1-indexes are stable structural indexes, which support a given query finite set of SAPE queries. A DL-1-index is constructed as follows:

- The initial DL-1-index is the coarsest stable structural index.
• Using DL1-Refine algorithm we refine the index step by step so that it supports a given finite set of SAPE queries.

Suppose

\[ T_i = \{ U \in V | U \cap T_G^i(R) \neq \emptyset \}, \]

\[ H_i = \text{Extent}(T_i) \cap \text{Label}^{-1}(S_i) \]

\((i = 0, \ldots, l - 1)\).

The DL1-Refine algorithm refines the index graph \( I_G \) to support a \( SAPE R = S_0 \ldots S_l \). It is shown as follows.

\[
\text{DL1-Refine}(I_G, R) \\
\text{begin} \\
1. \quad C \leftarrow \{ T_G^i(R) \} \\
2. \quad \text{for } i \leftarrow 0 \text{ to } l - 1 \text{ do} \\
3. \quad \quad C \leftarrow C \cup H_i \\
4. \quad \text{DL1-TOTALSplit}(I_G, C) \\
5. \quad C \leftarrow \emptyset \\
6. \quad \text{for each } U_0 \ldots U_l \text{ index node path matching } R, \text{ and } U_l \cap T_G(R) = \emptyset \text{ do} \\
7. \quad \quad \text{for } i \leftarrow 0 \text{ to } l - 1 \text{ do} \\
8. \quad \quad \quad C \leftarrow C \cup \{ U_i \cap \text{Label}^{-1}(S_i) \} \\
9. \quad \text{DL1-TOTALSplit}(I_G, C) \\
\text{end}
\]

\[
\text{DL1-TOTALSplit}(I_G, C) \\
\text{begin} \\
1. \quad \mathcal{P} \text{ is the partition over data nodes} \\
2. \quad \mathcal{P} \leftarrow \text{split}_\infty(\mathcal{P} \cup C) \\
3. \quad \text{for each edge } (u', u) \text{ do} \\
4. \quad \quad U, U' \text{ are index nodes such that } u \in U \text{ and } u' \in U' \\
5. \quad \quad \text{if } \not\exists \text{ an index edge from } U' \text{ to } U \text{ then} \\
6. \quad \quad \quad \text{Add an index edge from } U \text{ to } U' \\
\text{end}
\]

The refinement algorithm has two phases.

1. After the first phase (step 1-4.), we have:

   Suppose \( U \) is a target index node of \( R \) on \( I_G \) in which \( U \cap T_G(R) \neq \emptyset \). Then, \( U \) is sound.

2. After the second phase (step 5-9.), we have:

   There is no false target index node. It means: there does not exist an index node \( U \) of \( I_G \) such that \( U \) is a target index node of \( R \) and \( U \cap T_G(R) = \emptyset \).
Clearly, if the above conditions are satisfied \( I_G \), supports \( R \) after applying the DL1-Refine algorithm.

Let’s see the illustration for the \( DL-1 \)-index and the DL1-Refine algorithm which is shown in Figure 3.2. The data graph is represented in Figure 3.2(a). The 1-index and the data graph coincide. The initial \( DL-1 \)-index is represented in Figure 3.2(b). The \( DL-1 \)-index, which is constructed by refining the initial \( DL-1 \)-index to support \( R_1 = \sim^* (K|L) \) is represented in Figure 3.2(c). The \( DL-1 \)-index, which is constructed by refining the \( DL-1 \)-index in Figure 3.2(c) to support \( R_2 = \sim^* (B|C).E \) is represented in Figure 3.2(d).

In the refinement to support \( R_1 \), we refine the index graph by using \( C = \{\{1, 2\}\} \) at the first phase. The second phase is not necessary as there is no false target index node. In the refinement to support \( R_2 \), we refine the index graph by using \( C = \{\{5, 6\}, \{9, 10\}\} \) at the first phase. However, only the second phase changes the index graph to exclude the false target index node by using \( C = \{\{7\}\} \).

![Figure 3.2: An example of DL-1-indexes](image)

To prove the accuracy of the DL1-Refine algorithm we need the following corollary:

**Corollary 3.2.** Suppose \( P = \text{split}_\infty(C) \), and \( S_0, \ldots, S_l \) are sets of label values. Let \( C_0, \ldots, C_l \) be elements of \( C \), and \( U_0, \ldots, U_l \) be elements of \( P \) in which \( \text{label}(C_i) \subseteq S_i \) and \( U_i \cap C_i \neq \emptyset \ (i = 0, \ldots, l) \). Then, we have:

\[ U_i \subseteq C_i \text{ and } \text{label}(U_i) \subseteq S_i \text{ for each } i = 0, \ldots, l. \]

**Proof.** \( U_i \cap C_i \neq \emptyset \) implies that \( U_i \subseteq C_i \). Therefore, \( \text{label}(U_i) \subseteq \text{label}(C_i) \subseteq S_i \) for each \( i = 0, \ldots, l \). \( \square \)

**Proposition 3.9.** Suppose \( I_G \) and \( R \) are the structural index and the SAPE query which are used in DL1-Refine algorithm. Then, we have:

1. \( I_G \) supports \( R \) after applying the DL1-Refine algorithm.
2. Suppose at the beginning of the algorithm the 1-index is a refinement of \( I_G \). Then, after the refinement, the 1-index is still a refinement of \( I_G \).
Proof. 1. After the first phase (step 1-4.), suppose $U$ is a target index node of $R$ on $I_G$ in which $U \cap T_G(R) \neq \emptyset$. We will show that: $U$ is sound.

Since
i. $T_G(R)$ is an element of $C$ used in the splitting in step 4,
ii. $\text{label}(T_G(R)) \subseteq S_i$, and
iii. $U \cap T_G(R) \neq \emptyset$,
applying Corollary 5, we have:

$$U \subseteq T_G(R) \land \text{label}(U) \subseteq S_i.$$  

Let $u \in U \subseteq T_G(R)$ be a data node, $u_0 \ldots u_l$ be the data node path matching the end of $R$ ($u_l = u$), and $U_i$ be the index node which contains $u_i$ ($i = 0, \ldots, l - 1$). Since $u_i \in \text{label}^{-1}(S_i)$ and $u_i \in T_i$,

$$u_i \in H_i = \text{Extent}(T_i) \cap \text{label}^{-1}(S_i) \ (i = 0, 1, \ldots, l - 1)$$

Since $u_i \in H_i$ and $u_i \in U_i$,

$$U_i \cap H_i \neq \emptyset \ (i = 0, 1, \ldots, l - 1)$$

Since $\text{label}(H_i) \subseteq S_i$ and $H_i$ is an element of $C$ used in the splitting in step 4, applying Corollary 3.2 we have:

$$U_i \subseteq H_i \land \text{label}(U_i) \subseteq S_i(i = 0, \ldots, l - 1).$$

Hence, $U$ is sound.

We show that after the second phase (step 5-9.): there is no false target index node.

We assume that after step 9, there exists a target index node $U$ of $R$ and $U \cap T_G(R) = \emptyset$. Let $U_0 \ldots U_l$ be the index path node matching the end of $R$ ($U_l = U$), and $U'_0 \ldots U'_l$ be the index path node of $I_G$ after step 4 such that $U_i \subseteq U'_i$. Clearly, $U'_0 \ldots U'_l$ matches the end of $R$, and

$$U'_i \cap T_G(R) = \emptyset \lor U'_i \subseteq T_G(R).$$

Since $U \cap T_G(R) = \emptyset$ and $U \subseteq U'_i$, $U' \not\subseteq T_G$. Hence, $U'_i \cap T_G(R) = \emptyset$.

Therefore, $U'_0 \ldots U'_l$ satisfies the condition in step 6.

Let $K_i = U'_i \cap \text{label}^{-1}(S_i)$ ($i = 0, \ldots, l - 1$), which are elements of $C$ used in the splitting in step 9.

Since $U_i \subseteq U'_i$, we have:

$$U_i \cap K_i = U_i \cap U'_i \cap \text{label}^{-1}(S_i) = U_i \cap \text{label}^{-1}(S_i) \neq \emptyset \ (i = 0, \ldots, l - 1).$$

Applying Corollary 5 we have:

$$U_i \subseteq K_i \land \text{label}(U_i) \subseteq S_i \ (i = 0, \ldots, l - 1).$$

Since $\text{label}(U_i) \cap S_i \neq \emptyset$, there exists $u_i \in U_i$ such that $\text{label}(u_i) \in S_i$. 
Let \( u_0u_1 \ldots u_l \) be the data node path such that \( u_i \in U_i \) for each \( i = 0, \ldots, l \).

Since \( \text{label}(u_i) \in \text{label}(U_i) \subseteq S_i \) for each \( i = 0, \ldots, l - 1 \) and \( \text{label}(u_l) \in S_l \), \( u_0u_1 \ldots u_l \) matches the end of \( R \). Hence, \( u_l \in T_G(R) \), which violates the condition \( U \cap T_G(R) = \emptyset \).

Two above claims imply that \( I_G \) supports \( R \).

2. Obviously, if \( U, U' \) are the union of several index nodes of the 1-index then \( \text{Succ}(U), U \cap U' \) are too. Because \( T_G(R), \text{label}^{-1}(S_i) \) and \( \text{Extent}(T_i) \) are the union of several index nodes of the 1-index, so all elements of \( C \) are the union of several index nodes of the 1-index. Therefore, if at the beginning of the refinement the 1-index is a refinement of \( I_G \), after that the 1-index is still a refinement of \( I_G \).

The 1-index is a refinement of the initial \( DL-1 \)-index so the 1-index is a refinement of all \( DL-1 \)-indexes.

We can evaluate a \( SAPE \) on index graph by using the \textsc{Naive-Graph-Traversal} algorithm. In this case, the index graph takes the place of the data graph. The input set contains only the root of the index graph which simulate the root of the data graph. The output set is the set of index nodes. We also check if the current target index node is sound. If the index node is sound, the data nodes in its extent are added to the final result; otherwise, we validate these data nodes in the data graph.

3.4 \textbf{DL-A*}(k)-Index

3.4.1 \( A(k) \)-Index And \( k \)-Stable Structural Indexes

The foundation of \( A(k) \)-indexes [12] is \( k \)-\textit{similarity} relations.

\textbf{Definition 3.17.} (\( k \)-\textit{similarity}) The \( k \)-\textit{similarity} relations are defined inductively as follows:

1. Two data nodes \( u \) and \( v \) are 0-bisimilar \( (u \approx^0 v) \), if they have the same label.
2. Two data nodes \( u \) and \( v \) are \( k \)-bisimilar \( (u \approx^k v) \) if
   
   i. \( u \approx^{k-1} v \) and
   
   ii. for each edge \( (u', u) \) there exists an edge \( (v', v) \) such that \( u' \approx^{k-1} v' \), and vice versa.

They shown that the partition created by the \( k \)-\textit{similarity} is equal to \( \text{split}_k(P_L) \) [12]. This is also defined as the set of index nodes of the \( A(k) \)-index. An index edge is added from index node \( U' \) to index node \( U \) if and only if there exists an edge from \( u' \in U' \) to \( u \in U \). They also show that the \( A(k) \)-index is safe and sound with any \( SAPE \) which is not longer than \( k \).
However, the $A(k)$-index does not satisfy the first sufficient soundness condition as if $(U', U)$ is an index edge, $U$ may not be a subset of $\text{Succ}(U')$. We introduce and study on the $k$-stable structural indexes in which the first sufficient soundness condition is hold.

**Definition 3.18.** ($k$-stable structural index) Suppose $\mathcal{P}_0$, $\mathcal{P}_1$, ..., $\mathcal{P}_k$ are partitions in which $\mathcal{P}_{i+1} \preceq \text{split}(\mathcal{P}_i)$ for each $i = 0, \ldots, k - 1$. The $k$-stable structural index is determined by the $(\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_k)$ in which:

i. The set of index nodes is $V_i = \bigcup_{i=0}^{k} \mathcal{P}_i$.

ii. An index edge is added from $U'$ to $U$ if there exists an edge from $u' \in U'$ to $u \in U$ and there exists $i_0 \in \{0, 1, \ldots, k-1\}$ such that $U' \in \mathcal{P}_{i_0}$ and $U \in \mathcal{P}_{i_0+1}$.

$\mathcal{P}_0$, $\mathcal{P}_1$, ..., $\mathcal{P}_{k-1}$ are ancestor partitions. $\mathcal{P}_k$ is the main partition. If $U_0 U_1 \ldots U_l$ is an index node path, there exists $\mathcal{P}_{i_0}, \mathcal{P}_{i_0+1}, \ldots, \mathcal{P}_{i_0+l}$ in which $U_i \in \mathcal{P}_{i_0+i}$ for each $i = 0, 1, \ldots, l$. Therefore, all index node paths are not longer than $k + 1$. It implies that the $k$-stable structural indexes are acyclic and do not support any SAPE longer than $k$.

**Proposition 3.10.** The $k$-stable structural indexes satisfy the sufficient safeness condition and the first sufficient soundness condition for any SAPE which is not longer than $k$.

**Proof.** Let $u_0 u_1 \ldots u_l$ be a data node path ($l \leq k$), and $U_i \in \mathcal{P}_{i+k-1}$ be the index node in which $u_i \in U_i$ ($i = 0, 1, \ldots, l$). As $(u_i, u_{i+1})$ is a data edge so $(U_i, U_{i+1})$ is an index edge ($i = 0, 1, \ldots, l-1$). Hence, $U_0 U_1 \ldots U_l$ is an index node path and $u_0 u_1 \ldots u_l$ is simulated by $U_0 U_1 \ldots U_l$. It implies that the $k$-stable structural indexes satisfy the sufficient safeness condition for any SAPE which is not longer than $k$.

The condition $P_{i+1} \preceq \text{split}(P_i)$ and the definition of the index implies that if $(U', U)$ is an index edge then $U \subseteq \text{Succ}(U')$ (Proposition 3.5). Hence, the $k$-stable structural indexes satisfy the first sufficient soundness condition.

The $A(k)$-index stores information of all data node paths which are not longer than $k$ and only the $k$-bisimilarity partition is used for the query evaluation. With the $k$-stable structural indexes, we store information of the most important data node paths which are not longer than $k$ and determined by the set of given queries. By using the ancestor bisimilarity partitions the complexity of query evaluation is reduced.

To avoid refining the ancestor bisimilarity partitions, we modify the definition of target index node of a SAPE $R = S_0 S_1 \ldots S_l$ ($l \leq k$) on the $k$-stable structural index by a plus condition that the target index node must be an element of $\mathcal{P}_k$. Hence, if $U_0 U_1 \ldots U_l$ matches the end of $R$ and $U_i$ is a target node, we have:

$$U_i \in \mathcal{P}_{k-l+i} \quad (i = 0, \ldots, l).$$
Suppose $A_G$ is a $k$-stable structural index determined by $(P_0, P_1, \ldots, P_k)$, and $R$ is a SAPE, which is not longer than $k$. We say that $A_G$ supports $R$ if all target index nodes in $P_k$ are sound. Clearly, if $A_G$ supports $R$, it is safe and sound with $R$.

**Definition 3.19.** (Refinement relation of $k$-stable structural indexes) Suppose $A_G$ and $A'_G$ are $k$-stable structural index graphs which are determined by $(P_0, \ldots, P_k)$ and $(P'_0, \ldots, P'_k)$ respectively. $A_G$ is a refinement of $A'_G$ if $P_i$ is a refinement of $P'_i$ for each $i = 0, \ldots, k$.

The coarsest $k$-stable structural index is the $k$-stable structural index created by $(P_u, \text{split}(P_u), \ldots, \text{split}_k(P_u))$. Clearly, every $k$-stable structural index is a refinement of the coarsest $k$-stable structural index. Similarly to DL-1-indexes, the following proposition describes the inheritance of the support of a SAPE between a $k$-stable structural index and its refinements.

**Proposition 3.11.** Let $A'_G$ be a $k$-stable structural index graph, $A_G$ be a refinement of $A'_G$ and $R$ be a SAPE given by $S_0 \ldots S_l$ ($l \leq k$). Then, we have:

1. Suppose $U_0 \ldots U_l$ is an index node path of $A'_G$ matching $R$, and $U'_0, \ldots, U'_l$ are index nodes of $A'_G$ in which $U_i \subseteq U'_i$ ($0 \leq i \leq l$). Then, $U'_0 \ldots U'_l$ is also an index node path of $A'_G$ which matches $R$.

The proof of Proposition 3.11 is quite similar to the proof of Proposition 3.8.

### 3.4.2 DL-A*(k)-Index

The $DL-A^*(k)$-indexes are $k$-stable structural ones. They support a given finite set of SAPE queries which are not longer than $k$. A $DL-A^*(k)$-index is constructed as follows:

- The initial $DL-A^*(k)$-index is the coarsest $k$-stable structural index.
- Using AK-REFINE algorithm we refine the index step by step so that it supports a given finite set of SAPE queries.

Suppose $K_i = \{ U \in P_{k-l+i} | U \cap T^+_i(R) \neq \emptyset \}$. The AK-REFINE algorithm refining $A_G$ to support $R = S_0S_1 \ldots S_l$ ($l \leq k$) is shown as follows.

**AK-REFINE($A_G, R$)**

begin
1. $T_1 \leftarrow T^-_G(R)$
2. for $i \leftarrow 0$ to $l - 1$
3. $T_i \leftarrow \{ \text{Extent}(K_i) \cap label^{-1}(S_i) \}$
4. for $i \leftarrow 0$ to $l$
5. \text{AK-Split}(A_G; \{T_i\}, k - l + i)
6. \text{AK-Edge-Construct}(A_G)
7. \text{T}_i \leftarrow \emptyset \ (i = 0, \ldots, l - 1)
8. \text{for each } U_0 \ldots U_l \text{ index node path matching } R, \text{ and } U_l \cap T_G(R) = \emptyset \text{ do}
9. \quad \text{for } i \leftarrow 0 \text{ to } l - 1 \text{ do}
10. \quad \quad \text{T}_i \leftarrow T_i \cup (U_i \cap Label^{-1}(S_i))
11. \quad \text{for } i \leftarrow 0 \text{ to } l - 1 \text{ do}
12. \quad \quad \text{AK-Split}(A_G; \{T_i\}, k - l + i)
13. \quad \text{AK-Edge-Construct}(A_G)
end

AK-Split(A_G, C, i)
begin
1. \quad \mathcal{P}_i \leftarrow CRF(\mathcal{P}_i \cup C)
2. \quad \text{if } i < k \text{ then}
3. \quad \quad \text{AK-Split}(A_G, \mathcal{P}_i \cup \text{Succ}(\mathcal{P}_i), i + 1)
end

AK-Edge-Construct(A_G)
begin
1. \quad \text{for each edge } (u', u) \text{ do}
2. \quad \quad \text{for } i = 0 \text{ to } k - 1 \text{ do}
3. \quad \quad \quad \text{Let } U' \in \mathcal{P}_i, U \in \mathcal{P}_{i+1} \text{ be index nodes such that } u \in U \text{ and } u' \in U'
4. \quad \quad \quad \quad \text{if } \not\exists \text{ an index edge from } U' \text{ to } U \text{ then}
5. \quad \quad \quad \quad \quad \text{Add an index edge from } U' \text{ to } U
end

Similarly to the DL1-Refine algorithm, the AK-Refine algorithm also has two phases.

1. After the first phase (step 1-6.), if \( U \) is a target index node of \( R \) on \( I_G \) and \( U \cap T_G(R) \neq \emptyset \), \( U \) is sound with \( R \).

2. After the second phase (step 7-13.), there is no false target index node, thus \( A_G \) supports \( R \).

Let’s see the illustration for the DL-A*(k)-index and the AK-Refine algorithm which is shown in Figure 3.3. The data graph is represented in Figure 3.2(a). The A(1)-index and the data graph coincide. There are three DL-A*(1)-indexes. Each DL-A*(1)-index is determined by two partitions \( \mathcal{P}_0 = \{A_{00}\} \) and \( \mathcal{P}_1 = \{A_{11}\} \). We have \( \mathcal{P}_1 \succeq \mathcal{P}_0 \). Hence, for each index node \( A_{0i} \in \mathcal{P}_0 \), we enumerate the index nodes \( A_{1j} \in \mathcal{P}_1 \) which are subsets
Suppose $A_0$ on the right of that. For each index node $A_1 \in P_1$, we enumerate the data nodes which are in its extent on the right, and the its label on the left of this index node. The DL-A*(1)-index in Figure 3.3(b) is the result when we refine the initial DL-A*(1)-index in Figure 3.3(a) to support $R_1 = \ast (K|L)$, and the DL-A*(1)-index in Figure 3.3(c) is the result when we refine the DL-A*(1)-index in Figure 3.3(b) to support $R_2 = \ast (B|C).E$.

Suppose $A^*(k)$-index is the $k$-stable structural index which is determined by $(P_c, \ldots, split_k(P_c))$.

**Proposition 3.12.** Suppose $A_G$ and $R$ are the $k$-stable structural index and the SAPE query using in the AK-Refine algorithm. Then, we have:

1. $A_G$ supports $R$ after applying the AK-Refine algorithm.
2. Suppose at the beginning of the algorithm the $A^*(k)$ is a refinement of $A_G$. Then, the $A^*(k)$ is still a refinement of $A_G$ after the refinement.

**Proof.** The structure of the proof of Proposition 3.12 is similar to that of Proposition 3.9.

1. At first, we show that if $U$ is a target index node of $R$ on $I_G$ after the first phase such that $U \cap T_G(R) \neq \emptyset$, $U$ is sound.
   
   Because of calling $\text{AK-SPLIT}(A_G, \{T_G(R)\}, k)$ at step 5 and $U \cap T_G(R) \neq \emptyset$, we have:
   
   $U \subseteq T_G(R)$.

   Let $u$ be a data node in $U \subseteq T_G(R)$, $u_0 \ldots u_l$ be the data node path matching $R$ ($u_l = u$),
and \( U_i \in \mathcal{P}_k \) be index node which contains \( u_i \) \((i = 0, \ldots, l - 1)\). Let \( H_i = \text{Extent}(K_i) \cap Label^{-1}(S_i) \) \((i = 0, \ldots, l - 1)\). Obviously, \( H_i \cap U_i \neq \emptyset \) as \( u_i \in H_i \cap U_i \).

Because of calling \( \text{AK-Split}(A_G, \{H_i\}, k - l + i) \) at step 5 and \( H_i \cap U_i \neq \emptyset \), we have:

\[
U_i \subseteq H_i \quad (i = 0, \ldots, l - 1).
\]

Thus, \( \text{label}(U_i) \subseteq \text{label}(H_i) \subseteq S_i \) \((i = 0, \ldots, l - 1)\). Therefore, \( U \) is sound.

We assume that after step 13, there exists a target index node \( U \) of \( R \) such that \( U \cap T_{\mathcal{G}}(R) = \emptyset \).

Let \( U_0 \ldots U_i \) be the index node path matching \( R \) (\( U_i = U \)), and \( U'_0 \ldots U'_i \) be the index node path of \( I_G \) after step 6 such that \( U_i \subseteq U'_i \). Clearly, \( U'_0 \ldots U'_i \) matches \( R \), and

\[
U'_i \cap T_{\mathcal{G}}(R) = \emptyset \lor U'_i \subseteq T_{\mathcal{G}}(R).
\]

If \( U'_i \subseteq T_{\mathcal{G}}(R), U_i \subseteq T_{\mathcal{G}}(R) \). Hence, \( U'_i \cap T_{\mathcal{G}}(R) = \emptyset \). It implies that \( U'_0 \ldots U'_i \) satisfies the condition in step 8.

Let \( M_i = U'_i \cap \text{label}^{-1}(S_i), i = 0, \ldots, l - 1 \). Because of \( U_i \subseteq U'_i \), we have:

\[
U_i \cap M_i = U_i \cap \text{label}^{-1}(S_i) \neq \emptyset.
\]

Because of calling \( \text{AK-Split}(A_G, \{M_i\}, k - l + i) \) at step 12 and \( U_i \cap M_i \neq \emptyset \), we have:

\[
U_i \subseteq M_i \quad (i = 0, \ldots, l - 1).
\]

Thus, \( \text{label}(U_i) \subseteq \text{label}(M_i) \subseteq S_i \) \((i = 0, \ldots, l - 1)\). \( \text{label}(U_i) \cap S_i \neq \emptyset \) implies that there exists \( u_i \in U_L \) such that \( \text{label}(u_i) \subseteq S_i \).

Let \( u_0 u_1 \ldots u_l \) be the node path such that \( u_i \in U_i \) for each \( i = 0, \ldots, l \). \( \text{label}(u_i) \subseteq \text{label}(U_i) \subseteq S_i \) for each \( i = 0, \ldots, l - 1 \) and \( \text{label}(u_i) \in S_i \) imply that \( u_0 u_1 \ldots u_l \) matches \( R \) and \( u_l \in T_{\mathcal{G}}(R) \). It violates the condition \( U \cap T_{\mathcal{G}}(R) = \emptyset \).

Above claims implies that \( A_G \) supports \( R \).

2. Obviously, if \( U, U' \) are the unions of several elements of the \( \text{split}_i(P_c) \), \( U \cap U' \) is also the union of several elements of the \( \text{split}_i(P_c) \) and \( \text{Succ}(U) \) is the union of several elements of the \( \text{split}_{i+1}(P_c) \). (a)

Suppose \( C \) is the set of refining sets used in \( \text{AK-Split}(A_G, C, i) \) procedure. \( T_{\mathcal{G}}(R) \) is the union of several elements of \( \text{split}_k(P_c) \), and \( \text{label}^{-1}(S_i) \) and \( K_i \) are the union of several elements of \( \text{split}_{k-i}(P_c) \). (b)

(a) and (b) implies that all elements of \( C \) are the union of several elements of \( \text{split}_i(P_c) \).

Therefore, if at the beginning of the algorithm, the \( A^*(k) \) is a refinement of \( A_G \), the \( A^*(k) \) is still a refinement of \( A_G \) after the refinement.

\( \Box \)

The \( A^*(k) \) is a refinement of the initial DL-A*\((k)\)-index so the \( A^*(k) \) is a refinement of all DL-A*\((k)\)-indexes.

The DL-A*\((k)\)-indexes only support \textit{SAPE} queries which are not longer than \( k \). In the case the length of the path expression is longer than \( k \), the information supplied by index
edges of the DL-A*(k)-index is not enough. We suggest the DL-A*(k)-index should be extended by adding extended index edges between the index nodes in k-layer (the dotted lines in Figure 3.3). We add an extended index edge from $U' \in P_k$ to $U \in P_k$ if and only if there exists a data edge from $u' \in U'$ to $u \in U$.

In summary, when the path expression is not longer than $k$, the evaluation is calculated on origin index edges. In the case that the path expression is longer than $k$, the first $k$ index edges of the matching paths are found out by using the origin index edges. The remainders of these paths are found out by using the extended index edges.

### 3.5 Summary

We have studied on the mathematical background, an overview of structural indexes, SAPE queries and the dynamic labeling techniques for structural indexes.

Our mathematical background is based on the split, refine and CRF operations of partitions. It is used to model, create and refine the structural indexes. SAPE is a general type of the near primary queries and designed for supporting wild- or alternation cards. The classical static labeling indexes do not well support SAPE queries. We have shown the sufficient conditions of the safeness and soundness of a structural index with a SAPE. The dynamic labeling techniques based on our mathematical background not only make the size of an index graph smaller, but also support the SAPE queries better. Furthermore, they make the index graph more dynamic and adaptive. Every static labeling structural index can be improved so that it supports dynamic labeling.

The DL-1-index is developed from the 1-index to support dynamic labeling. The stable structural indexes and their properties are introduced and studied. The 1-index is only a special case of stable structural indexes. The DL-1-indexes are the stable structural indexes which support a finite set of SAPE queries. The sizes of the DL-1-indexes are bound by the size of the 1-index. In the experiments in Appendix A, the size of the DL-1-index is about 81% of that of the 1-index, and the average processing cost of SAPE queries over the DL-1-index is about 82% of that over the 1-index.

The DL-A*(k)-index is developed from the A(k)-index. By storing ancestor bisimilarity layers we reduce the cost of the query evaluation on the DL-A*(k)-index. In the experiments in Appendix A, the size of the DL-A*(k)-index is about 47% of that of the
A(k)-index, and the average processing cost of SAPE queries over the DL-A*(k)-index is about 7% of that over the A(k)-index. With those results, we believe that the DL-A*(k)-index is one of the most efficient and adaptive structural indexes.
Chapter 4

Efficient Processing Of Primary Queries Over Near Tree XML Data Sets

There are many works [3, 9, 14, 15, 25] proposed for indexing the basic tree structure to improve the speed of query processing over XML data sets. In those works, only the tree structure of XML data set is considered, the reference edges are ignored. The indexes over the basic tree help us to find the ancestor-descendant relationship and determine the label path between two arbitrary data nodes in constant time. The processing cost of primary queries over the XML tree with these indexes only depends on the size of the input set and output set.

In this chapter, our data sets are near tree XML data sets, and only primary queries are considered. The number of reference edges of the near tree XML data sets is much smaller than the number of edges. In chapter 2, we show that every regular query can be decomposed into primary queries. Hence, the regular query processing problem can be reduced to the primary query processing problem.

Our algorithms for efficient processing of primary queries over the near tree XML data sets take the advantages of the structural- and tree structure indexes. The data graph is considered as the union of two components: the basic tree and the link graph. The result of the primary query on the data graph can be considered as the union and concatenation of two sub-results of this query on the link graph and the basic tree. The sizes of the link graph are linear functions of the number of reference edges, which is much smaller than that of the data graph. Since the basic tree is indexed and the size of the link graph is
small, our algorithm is more efficient than the naive algorithm.

The strong 1-index, in which the basic tree structure is conserved, is studied. The corresponding efficient primary query processing algorithm uses the tree structure index on the strong 1-index. The complexities of the algorithms constructing the tree structure index and the link graph are linear functions of edges. The new primary query processing algorithm is more efficient than the existing approaches when we apply tree structure index or structural index separately.

The chapter is organized as follows. Section 4.1 is the preliminary. Basic concepts, definitions and the efficient primary query processing algorithm over XML trees with the tree structure indexes are introduced. Section 4.2 is about the efficient primary query processing algorithm over the near tree XML data sets. The link graph is introduced and studied on in this section. In section 4.3, the strong 1-index and the corresponding primary query processing algorithm is investigated on. Section 4.4 discusses on the algorithms for constructing the indexes. Section 4.5 gives the conclusion of this chapter.

4.1 Preliminaries

There are two kinds of data edges: basic edges \((E_b)\) and reference edges \((E_f)\). The basic tree of data graph \(G = (V, E, \Sigma, r, label)\) is \(T = (V, E_b, \Sigma, r, label)\) created by ignoring reference edges. \(G\) is a tree if \(E_f = \emptyset\), and is near tree if \(|E_f| \ll |E_b|\).

A data node path is a basic path if it contains only basic edges. Otherwise it is a complex path if it contains at least one reference edge. Data node \(u\) is an ancestor of data node \(v\) if \(u \neq v\) and there exists a basic path from \(u\) to \(v\). \(v\) is a descendant of \(u\) if \(u\) is an ancestor of \(v\). In this case there is one and only one basic path connecting from \(u\) to \(v\). The root \(r\) is an ancestor of any data node except itself.

\(u\) is a source reference node if there is a reference edge from \(u\) to some data node, and \(u\) is a destination reference node if there is a reference edge from some data node to \(u\). \(u\) is reference node if it is a source reference node or destination reference node. \(V_f, S_f\) and \(D_f\) denote for the set of reference nodes, source reference nodes and destination reference node respectively. Certainly, we have:

\[S_f := \{u | \exists (u, v) \in E_f\},\]
\[
D_f := \{v | \exists (u, v) \in E_f \}, \\
V_f := S_f \cup D_f.
\]

### 4.1.1 Tree Structure Indexes

Suppose \(a, b \in \mathbb{R}\) and \(a \leq b\). We define:

\[
[a, b] := \{x | a \leq x \leq b\}
\]

Suppose \(f, g: V \mapsto \mathbb{N}\) are functions mapping from \(V\) to \(\mathbb{N}\). \((f, g)\) is called AD system if following conditions are satisfied:

i. \(\forall u \in V : f(u) \leq g(u)\).

ii. \(\forall v, u \in V, u \neq v : f(u) \neq f(v)\).

iii. \(\forall v, u \in V, u \neq v : \) if \(u\) is an ancestor of \(v\), we have:

\[
[f(v), g(v)] \subseteq [f(u), g(u)].
\]

iv. \(\forall v, u \in V, u \neq v : \) if \(u\) is not an ancestor of \(v\) and vise versa, we have:

\[
[f(u), g(u)] \cap [f(v), g(v)] = \emptyset.
\]

Suppose \((f, g)\) is an AD system. Certainly, if the each node \(u\) is indexed by \((f(u), g(u))\), the ancestor-descendant relationship between two arbitrary nodes \(u\) and \(v\) can be checked in constant time. A simple AD system can be constructed by choosing \(f\) as the position of the data node and \(g\) as the position of the last descendent or itself node of the data node in the pre-order sequence of the basic tree. The pre-order sequence of the basic tree shown in Figure 4.1 is 1 2 4 8 13 9 14 3 5 10 15 6 11 16 7 12 17. Hence \(f(4) = 3\) and \(g(4) = f(14) = 7\) as node 14 is the last descendent of node 4 in the pre-order sequence. The \((f, g)\) values of the other nodes can be found at the table of Figure 4.1.

**Definition 4.1.** (Tree structure index)

\((f, g, lr)\) is a tree structure index if \((f, g)\) is an AD system and \(lr(u)\) is the label of the basic path from the root to \(u\).

Suppose \(\alpha = a_1a_2 \ldots a_l \in \Sigma^*\) and \(l = \text{length}(\alpha) > 0\). Then, we define:

\[
\text{prefix}_0(\alpha) := \epsilon, \quad \text{prefix}_i(\alpha) := a_1a_2 \ldots a_i,
\]
Figure 4.1: An example of tree structure index

\[
\text{suffix}_0(\alpha) := \epsilon, \quad \text{suffix}_i(\alpha) := a_{i-I}a_{i+2}\ldots a_l,
\]

\[
\text{substr}_{l,0}(\alpha) := \epsilon, \quad \text{substr}_{l,k}(\alpha) := a_{t+1}\ldots a_{t+k-1},
\]
in which \(1 \leq i \leq l\) and \(1 \leq t \leq l+1-t\).

prefix_i(\alpha), \text{suffix}_i(\alpha)\) are the prefix, the suffix of \(\alpha\) of which the length is \(i\) respectively. \(\text{substr}_{l,k}(\alpha)\) is the sub-string of \(\alpha\) at position \(t\) and the length is \(k\).

**Proposition 4.1.** Suppose \(\alpha \in \Sigma^*,\ l = \text{length}(\alpha) > 0\) and \((f, g, lr)\) is a tree structure index. In the basic tree \(T\), \(u \xrightarrow{R} v\) if and only if following condition is satisfied:

\[
f(u) \leq f(v) \leq g(u) \land lr(v) = lr(u).\text{suffix}_{l-1}(\alpha) \land \text{prefix}_1(\alpha) = \text{label}(u)
\]

**Proof.** Since \((f, g)\) is an AD system, there exists a basic path from \(u\) to \(v\) if and only if:

\[
f(u) \leq f(v) \leq g(u) \quad (a).
\]

Suppose there is a basic path \(p\) from \(u\) to \(v\). The definition of \(lr\) implies that \(\alpha = \text{label}(p)\) if and only if:

\[
\text{lr}(v) = lr(u).\text{suffix}_{\text{length}(\alpha)-1}(\alpha) \land \text{label}(u) = \text{prefix}_1(\alpha) \quad (b).
\]

(a) and (b) imply the accuracy of the preposition. \(\square\)
Proposition 4.1 is the foundation of efficient processing primary queries over the XML trees \((E_f = \emptyset)\). In this context, each \(\alpha \in \Sigma^*\) is coded by a numeric value as \(\Sigma\) is finite \([P1]\). The necessary data structures are used (for example B+ tree \([25]\)) for storing and processing the tree structure index. These enable us to check the conditions in Proposition 4.1 with low cost. In reality, the cost of processing over these indexes are very small when comparing with that over the data graph. Hence, \textit{if the cost of processing over the tree structure indexes is ignored, the cost of processing a primary query over the XML tree depends only on the sizes of the input set and output set.}

4.2 Link Graph And Improved Algorithm

The data graph can be considered as the union of the basic tree and the link graph. In the case of the near tree XML data sets, the link graph is very small when comparing with the size of data graph. The basic paths matching the primary query are determined by processing over the basic tree. The complex paths matching the query are determined by processing over the link graph, the basic tree and the complex representation of the primary query graph. The link graph is introduced and studied on in subsection 4.2.1. Subsection 4.2.2 is about the complex representation of primary queries. The improved algorithm is investigated in subsection 4.2.3.

4.2.1 Link Graph

The reference node \(u\) is the \textit{closest reference ancestor} of data node \(v\) if \(u\) is ancestor of \(v\) and there does not exist any reference node \(w\) such that \(w\) is an ancestor of \(v\) and \(w\) is a descendent of \(u\). \(u \rightsquigarrow v\) denotes that \(u\) is the closest reference ancestor of \(v\). In the case \(u \rightsquigarrow v, u\) is the \textit{closest reference ancestor type I} of \(v\) denoted by \(u \rightsquigarrow I v\) if \((u,v)\) is a basic edge, otherwise \(u\) is the \textit{closest reference ancestor type II} of \(v\) denoted by \(u \rightsquigarrow II v\). The basic path from \(u\) to \(v\) is a \textit{basic link path} if \(u, v\) are reference nodes and \(u \rightsquigarrow v\).

**Definition 4.2.** (Link graph) The link graph \(L = (V_L, E_L, \Sigma, label_1, label_2)\) of data graph \(G = (V, E, \Sigma, r, label)\) is a labeled directed graph in which:

i. \(V_L = V_f \cup V_{sym}\) is the set of nodes of \(L\). \(V_f\) is the set of reference nodes of \(G\). \(V_{sym}\) is the set of symbol nodes.

ii. \(E_L = E_f \cup E_{sym}\), the set of the edges in \(L\). \(E_f\) is the set of the reference edges of \(G\). \(E_{sym}\) is the set of symbol edges.
iii. $\text{label}_1 : V_f \mapsto \Sigma$, $\text{label}_2 : V_{\text{sym}} \mapsto \Sigma^*$ are labeling functions. $\forall u \in V_f : \text{label}_1(u) = \text{label}(u)$.

$\text{label}_2$, $E_{\text{sym}}$ and $V_{\text{sym}}$ are determined as follows:

iv. For each $u, v \in V_f$ and $u \sim_I v$, a symbol edge $e \in E_{\text{sym}}$ from $u$ to $v$ is added to symbolize the basic edge $(u, v)$.

v. For each $u, v \in V_f$ and $u \sim_{II} v$, a symbol node $w \in V_{\text{sym}}$ and two symbol edges $e_1, e_2 \in E_{\text{sym}}$ are added to symbolize the basic path from $u$ to $v$, in which $e_1$ is from $u$ to $w$, $e_2$ is from $w$ to $v$ and $w$ is labeled by $\alpha = \text{label}_2(w)$ such that label($u$) $\cdot$ label($v$) is the label of the basic path from $u$ to $v$.

![Diagram](image)

Figure 4.2: Link graph of the data graph shown in Figure 4.1

Let’s see the example of link graph shown in Figure 4.2. It is the link graph of the data graph shown in Figure 4.1. We have: $V_f = \{3, 5, 15, 16\}$, $3 \sim_I 5$, $5 \sim_{II} 15$ and $3 \sim_{II} 16$. The basic link path 3.5 is symbolized by edge $(3, 5)$ in the link graph. The basic link paths from 5.10.15 and 3.6.11.16 are symbolized by the paths 5.17.15) and 3.18.16 in the link graph respectively in which 17, 18 are symbol nodes and $\text{label}_2(17) = e$ and $\text{label}_2(18) = c.f$.

**Proposition 4.2.** Let $\mathcal{L}$ be the link graph of the data graph $\mathcal{G}$, and $u$, $v$ are reference nodes ($u, v \in V_f$). Then, there exists a path on $\mathcal{G}$ from $u$ to $v$ of which label is $\alpha$ if and only if there exists a path on $\mathcal{L}$ from $u$ to $v$ of which label is $\alpha$.

**Proof.** Let $p$ be a data node path from $u \in V_f$ to $v \in V_f$ and $u_0, u_1, \ldots, u_n$ be the sequence of reference nodes of $p$ ordered by their position in $p$ in which $u_0 = u$ and $u_n = v$. Clearly, the sub-path $p_i$ of $p$ from $u_i$ to $u_{i+1}$ ($i = 0, 1, \ldots, n - 1$) is a reference edge or basic link path. Let $p_L = \text{symbol}(p)$ be a path on $\mathcal{L}$ from $u$ to $v$ and containing $u_0, u_1, \ldots, u_n$ in which the sub-path $p'_L$ from $u_i$ to $u_{i+1}$ is:

1. reference edge $(u_i, u_{i+1})$ if $p_i$ is the reference edge $(u_i, u_{i+1})$, or
2. symbol edge $(u_i, u_{i+1})$ if $p_i$ is the basic edge $(u_i, u_{i+1})$ $\sim_I u_{i+1}$, or
3. path $u_i \cdot w \cdot u_{i+1}$ symbolizing $p_i$ if $p_i$ is the basic link path and $u_i \sim_{II} u_{i+1}$.

Since $\text{label}(p_i) = \text{label}(p'_L)$ for each $i = 0, 1, \ldots, n - 1$, we have $\text{label}(p) = \text{label}(p_L)$. Clearly, the symbol function is bijective since all basic link paths and reference edges are
symbolized by one and only one path of the link graph. Hence the proposition is true.

\[ \square \]

**Proposition 4.3.** The sizes of the link graph are the linear functions of the number of reference edges of its data graph. Precisely, \( |V_L| \leq 4|E_f| \), and \( |E_L| \leq 5|E_f| \).

**Proof.** We have: \( |S_f| \leq |E_f| \) and \( |D_f| \leq |E_f| \). Hence,

\[
|V_f| = |S_f \cup D_f| \leq |S_f| + |D_f| \leq 2|E_f|
\]

Let \( H := \{(u, v) \in V_f \times V_f | u \sim v\} \), \( H_1 := \{(u, v) \in H | u \sim_I v\} \) and \( H_2 := \{(u, v) \in H | u \sim_{II} v\} \). The definition of link graph implies that

\[
|V_{sym}| = |H_2| \leq |H|,
\]

\[
|E_{sym}| = 2|H_2| + |H_1| = |H| + |H_2| \leq 2|H|.
\]

For each \( v \in V_f \), there is maximum one reference node \( u \) such that \( u \sim v \). Hence,

\[
|H| = |H_1| + |H_2| \leq |V_f| \leq 2|E_f|.
\]

Finally,

\[
|V_L| \leq |V_f| + |V_{sym}| \leq |V_f| + |H| \leq 2|V_f| \leq 4|E_f|,
\]

and

\[
|E_L| = |E_f| + |E_{sym}| \leq |E_f| + 2|H| \leq 5|E_f|.
\]

\[ \square \]

### 4.2.2 Complex Representation Of Primary Queries

Each edge of the query graph is labeled by a character. Thus, when a word is is checked if it matches the query or not, the word must be read character by character. The complex representation of a query is designed for the case that the checked word is read by a sequence of characters.

Suppose \( \alpha = a_1a_2\ldots a_l \) (\( a_i \in \Sigma, i = 1, 2, \ldots, l \)) and \( l = \text{length}(\alpha) > 0 \). The query graph of \( R_\alpha \) is \( Q_\alpha = (V_\alpha, E_\alpha, \Sigma, q_0, \{q_l\}) \) in which: \( V_\alpha = \{q_0, q_1, \ldots, q_l\} \); \( E_\alpha = \{(q_i, a_{i+1}, q_{i+1}) | i = 0, 1, \ldots, l - 1\} \); \( q_0 \) is the start state and \( q_l \) is the final state. Clearly, the corresponding automata of \( R_\alpha \) changes from the state \( q_i \) to \( q_j \) (\( i < j \)) if and only if the input string is \( \text{str}_{i+1,j-1}(\alpha) \).

**Definition 4.3.** (Complex representation of primary queries)

Suppose \( Q_\alpha = (V_\alpha, E_\alpha, \Sigma, q_0, \{q_l\}) \) is the query graph of the primary query \( R_\alpha \). The complex representation of \( R_\alpha \) is \( Q^c_\alpha = (V_\alpha, E^c_\alpha, \Sigma, q_0, \{q_l\}) \) in which

\[
E^c_\alpha = \{(q_i, \text{str}_{i+1,j-1}(\alpha), q_j) | 0 \leq i < j \leq l\}.
\]
Proposition 4.4. The labels of all path of the complex representation of primary query $R_\alpha$ from state $q_i$ to state $q_j$ have the same value which is equal to $\text{substr}_{i+1,j-i}(\alpha)$ ($0 \leq i < j \leq l$).

Proof. Let $p = q_{t_0}q_{t_1} \ldots q_{t_l}$ be an arbitrary path of the complex representation of $R_\alpha$ from state $q_i$ to state $q_j$ in which $t_0 = i$ and $t_l = j$. We have:

\[
\text{label}(p) = \text{label}(q_{t_0}q_{t_1}) \cdot \text{label}(q_{t_1}q_{t_2}) \ldots \text{label}(q_{t_{l-1}}q_{t_l}) \\
= \text{substr}_{t_0+1,t_1-t_0}(\alpha) \cdot \text{substr}_{t_1+1,t_2-t_1}(\alpha) \ldots \text{substr}_{t_{l-1}+1,t_l-t_{l-1}}(\alpha) \\
= \text{substr}_{i+1,j-i}(\alpha)
\]

\[
\square
\]

4.2.3 Improved Algorithm

In the case of the naive graph traversal algorithm, the whole data graph is traversed. With the improved algorithm, only the link graph is traversed and the other computations are processed over the tree structure index. The improved algorithm works as follows:

Suppose the primary query is $R_\alpha$ ($\alpha \in \Sigma^*$), the input set is $I$, the output set is $O$ and $l = \text{length}(\alpha)$. Consider the basic paths and the complex paths matching the given primary query.

Basic paths. The set of basic paths matching the given primary query are determined by processing over the tree structure index as we introduced and discussed in the previous section.

Complex paths. We split a complex path $p$ from $u$ to $v$ matching the given primary query into three sections:

1. The first section is from $u$ to $w \in S_f$ which is the source node of the first reference edge of $p$;
2. The second section from $w$ to $t \in D_f$ which is the destination node of the last reference reference edge of $p$;
3. The third section is from $t$ to $v$.

The first and third sections are basic paths. Their labels are the prefixes (in the case of the first sections) or suffixes (in the case of the third sections) of $\alpha$. These sub-paths are determined them by processing over the tree structure index. The second section can be determined by processing over the link graph.
We define:

\[ R_1 := \{(u, w, i) \in I \times S_f \times \mathbb{N} | 1 \leq i \leq l - 1 \land \beta = \text{prefix}_i(\alpha) \land u \xrightarrow{R_3} T w\}, \]

\[ R_3 := \{(t, v, j) \in D_f \times O \times \mathbb{N} | 1 \leq j \leq l - 1 \land \beta = \text{suffix}_j(\alpha) \land t \xrightarrow{R_3} T v\}, \]

\[ R_2 := \{(w, i, t, j) \in S_f \times \mathbb{N} \times D_f \times \mathbb{N} | 1 \leq i, j \leq l - 1 \land i + j < l \land (w, i) \in R_1 \land (t, j) \in R_3 \land \beta = \text{substr}_{i,l-i-j}(\alpha) \land w \xrightarrow{R_3} G t\}. \]

Clearly, the first-, second- and third sections which are defined above are the elements of \( R_1, R_2 \) and \( R_3 \) respectively. If \( R_1, R_2 \) and \( R_3 \) are determined, the set of pairs \((u, v)\) in which there exists a complex path from \( u \) to \( v \) matching \( R_{\alpha} \) is determined as

\[ R_c = \{(u, v) | \exists w, i, j, t : (u, w, i) \in R_1 \land (w, i, t, j) \in R_2 \land (t, j, v) \in R_3\}. \]

\( R_1 (R_3) \) is determined by finding the results of the primary queries \( \text{prefix}_i(\alpha) (\text{suffix}_j(\alpha)) \) over the basic tree with the input set \( I (D_f) \) and the output set \( S_f (O) \). Moreover, \(|D_f|\) and \(|S_f|\) are bound by the number of reference edges, \(|E_f|\). Hence, if the costs of processing over the tree structure indexes are ignored, the costs of determining \( R_1 \) and \( R_3 \) depend only on the sizes of the input set, the output set and the number of reference edges.

If \( R_1 \) and \( R_3 \) are determined, \( R_2 \) is determined by traversing the link graph and the complex representation of the query graph as follows:

Suppose \( L \) is the link graph, \( Q^c \) is a complex representation of primary query \( R_{\alpha} \),

\[ I := \{(w, q_i) | (u, w, i) \in R_1\}, \]

\[ O := \{(t, q_j) | (v, t, j) \in R_3\}, \]

and

\[ E^c_R(q, \text{label}(u)) = \{p \in V_\alpha | (q, \text{label}(u), p) \in E_\alpha\}. \]

The \textsc{Link-Graph-Traversal} procedure returns \( R_2 \).

\begin{verbatim}
LINK-GRAPH-TRAVERSAL(\mathcal{L}, Q^c, I, O) \hspace{1cm} SCAN(u,q)
begin
1. Result ← \emptyset
2. for each \((u', q') \in I\) do
3. \hspace{1cm} Visited ← Visited \cup (u, q)
4. for each \(p \in E^c_R(q, \text{label}(u))\) do
5. \hspace{2cm} if \((u, p) \in O\) then
6. \hspace{3cm} Result ← Result \cup \{(u, p)\}

\end{verbatim}
4. Scan \((u', q')\)
5. return Result

The Link-Graph-Traversal procedure works quite similarly to the Naive-Graph-Traversal procedure introduced in chapter 2. In the Naive-Graph-Traversal procedure, each tour begins at a pair of data node and state node which is element of \(I \times \{q_0\}\) and the result is extended by checking the condition \((u, p) \in O \times F\). In the Link-Graph-Traversal procedure, each tour begins at a pair of data node and state node which is element of \(I\), the result is extended by checking the condition if \((u, p) \in O\). \(I\) and \(O\) are the projections of \(R_1\) and \(R_3\) respectively.

**Proposition 4.5.** The Link-Graph-Traversal\((L, Q^c, I, O)\) algorithm stops after finite steps and returns exactly \(R_2\).

**Proof.** The proof of this proposition is similar to the proof of Proposition 2.1, which is about the accuracy of the Naive-Graph-Traversal algorithm.

Clearly, the complexity of the Link-Graph-Traversal algorithm depends only the link graph. Hence, the complexity of the improved algorithm depends only the link graph in the case the cost of processing over the tree structure is ignored.

### 4.3 Strong 1-Index

The strong 1-index is improved from the 1-index, which is introduced in section 3.3, to keep the basic tree of the data graph. The strong 1-index is based on the strong bisimilarity relation which is considered as a special case of bisimilarity.

**Definition 4.4.** (Strong bisimilarity)

Two data nodes \(u\) and \(v\) are strong bisimilar \((u \approx_s v)\), if:

i. They have the same label.

ii. If \((u', u)\) is a basic edge, then there exists \(v'\) such that \(u' \approx_s v'\) and \((v', v)\) is also a basic edge, and vice versa.

iii. If \((u', u)\) is a reference edge, then there exists \(v'\) such that \(u' \approx_s v'\) and \((v', v)\) is also a reference edge, and vice versa.

**Remark 4.1.** A strong bisimilarity relation is also a bisimilarity relation.
The remark is implied directly by the definitions of the strong bisimilarity- and the bisimilarity relations.

The definition of the $\text{split}_\infty$ operation introduced in chapter 3 uses $E \subseteq V \times V$ relation for determining the $\text{split}$ and $\text{Succ}$ functions. Thus, We say that the result is stable over $\text{Succ}$ function. Suppose $\mathcal{B}$ is the partition of a strong bisimilarity relation. The first condition of Definition 4.4 implies that $\mathcal{B}$ is a refinement of $\mathcal{P}_c$. The second and the third conditions implies that $\mathcal{B}$ is stable over $\text{Succ}_b$ and $\text{Succ}_f$. It means for all $B, B' \in \mathcal{B}$, we have:

$$\text{Succ}_b(B) \cap B' = \emptyset \lor B' \subseteq \text{Succ}_b(B)$$

and

$$\text{Succ}_f(B) \cap B' = \emptyset \lor B' \subseteq \text{Succ}_f(B)$$

The coarsest stable partition over $\text{Succ}_b$ and $\text{Succ}_f$, $\mathcal{B}_s$, can be computed as follows:

1. Compute $\mathcal{B}_1 = \text{split}_\infty(\mathcal{P}_c)$ by replacing $E$ by $E_b$.

2. Compute $\mathcal{B}_s = \text{split}_\infty(\mathcal{B}_1)$ by replacing $E$ by $E_f$.

The accuracy of the algorithm implies by Proposition 12.

**Corollary 4.1.** $\mathcal{B}_s$ is a stable partition.

**Proof.** Suppose $B, B' \in \mathcal{B}_s$ and $\text{Succ}(B) \cap B' \neq \emptyset$. Since $\text{Succ}(B) = \text{Succ}_f(B) \cup \text{Succ}_b(B)$ so

$$B' \cap \text{Succ}_f(B) \neq \emptyset \lor B' \cap \text{Succ}_b(B) \neq \emptyset$$

Since $B' \cap \text{Succ}_f(B) \neq \emptyset \Rightarrow B' \subseteq \text{Succ}_f(B)$ and $B' \cap \text{Succ}_b(B) \neq \emptyset \Rightarrow B' \subseteq \text{Succ}_b(B)$ so

$$B' \subseteq \text{Succ}_f(B) \lor B' \subseteq \text{Succ}_b(B)$$

It implies that $B' \subseteq \text{Succ}(B)$. Therefore, $\mathcal{B}_s$ is a stable partition. \qed

**Definition 4.5.** *(The strong 1-index)*

The strong 1-index $I_G = (V_I, E_I, \Sigma, r_I)$ is the structural index in which $V_I = \mathcal{B}_s$. Furthermore, an index edge is a (basic/reference) index edge if it simulates a (basic/reference) edge. In the case an index edge simulates some basic edge and reference edge, we replace this index edge by a basic edge simulating basic edges and a reference edge simulating reference edges.
An example of the strong 1-index is shown in Figure 4.3. In the example, there are two index edges from index node \{2,3\} to index node \{5\}. The first edge is a basic index edge simulating basic data edges. The second edge is a reference index edge simulating reference edges. Each index edge can not be both of basic index edge and reference edge.

Similar to the 1-index, all data nodes in the same index node of the strong 1-index have the same label and \(B_s\) is a stable partition so the strong 1-index is safe and sound with any primary queries.

**Proposition 4.6.** The basic edges of the strong 1-index create a tree, and the root of this tree is the index node containing only the root of the data graph. Furthermore, all data nodes in an index node have the same lr value which equals to the label of the basic index node path from the root to this index node.

**Proof.** Since the root has no incoming edge so the root, \(r\), only strong bisimilar with itself. Thus, the index node simulating the root, \(I_{\text{root}}\), has only one element (\(I_{\text{root}} = \{r\}\)).

Clearly, each data node, basic data edge and reference data edge is simulated by one and only one index node, basic index edge and reference index edge respectively. Therefore, each data node path is simulated by one and only one index node path. We will show that for each index node \(I\) there is one and only one basic index node path from \(I_{\text{root}}\) to \(I\).

Let \(I\) be an index node, \(u\) be a data node such that \(u \in I\). The basic path from \(r\) to \(u\) is \(p = u_0u_1\ldots u_k\) in which \(u_0 = r\) and \(u_k = u\). Suppose \(p_I = U_0U_1\ldots U_k\) is a index node path simulating \(p\). Clearly, \(U_0 = I_{\text{root}}\) and \(U_k = I\). Because \((u_i, u_{i+1})\) is simulated by \((U_i, U_{i+1})\) and \((u_i, u_{i+1})\) is a basic edge, \((U_i, U_{i+1})\) is a basic index node. Hence, \(p_I\) is the basic node path from \(I_{\text{root}}\) to \(I\). It implies that for each index node \(I\) there exists a basic index node path from \(I_{\text{root}}\) to \(I\).

Suppose \(p_I = U_0U_1\ldots U_k\) is a basic index node path from \(I_{\text{root}}\) of to \(I\) (\(U_0 = I_{\text{root}}\) and \(U_k = I\)). Since \(U_{i+1} \subseteq \text{Succ}\_b(U_i)\) for each \(i = 0, 1, \ldots, k - 1\), for each \(u \in I = U_k\) there
exists a basic path \( p = u_0u_1 \ldots u_k \) in which \( u_i \in U_i \) \((i = 0, 1, \ldots, k)\) and \( u_k = u \). Since \( U_0 = \{r\}, \ u_0 = r \). (1)

Suppose there exists two different basic index node paths from \( I_{\text{root}} \) to \( I \). (1) implies that for each \( u \in I \) there exists two different basic paths from \( r \) to \( u \). This is not true as basic data edges create a tree. Hence, there is one and only one basic path from \( I_{\text{root}} \) to every index node. Therefore, the basic index edges create a tree and the root of this tree is \( I_{\text{root}} \).

Finally, if basic path \( p = u_0u_1 \ldots u_k \) is simulated by basic index node path \( p_I = U_0U_1 \ldots U_k \), \( p \) and \( p_I \) have the same label. Therefore, all data nodes in an index node have the same \( lr \) value and it equals to the label of the basic index node path from the root to this index node.

\[ \square \]

The strong 1-index is safe and sound with any primary queries. Furthermore, the strong 1-index keeps the basic tree structure of the data graph. Hence, the tree structure indexes can be build over the strong 1-index for processing the primary queries. This approach combines the strong points of the structural indexes and the tree structured indexes. The experiments shown in Appendix B confirm the efficiency of this algorithm.

### 4.4 Index Constructions

In this section, we introduce the algorithms for constructing the tree structure indexes over the strong 1-index and the data graph.

#### 4.4.1 Tree Structure Index Construction

A tree structure index \((f, g, lr)\) can be created by traverse the given tree with pre-order sequence of nodes. \( f \) is defined as the position of the node in the pre-order sequence. \( g \) is defined as the position of the last descendent node in the pre-order sequence. \( lr \) is computed from the \( lr \) of the parent. The algorithm is shown below:

\[
\text{Tree-Structure-Index}(T) \quad \text{Build}(I, parentLr)
\]

\[
\begin{align*}
\text{begin} & \quad \text{build} \\
1 & \quad \text{curF} \leftarrow 1 \\
2 & \quad \text{Build}(\text{Root}(T), \varepsilon) \\
\text{end} & \quad 1I.lr \leftarrow parentLr.label(I) \\
& \quad 2I.f \leftarrow curF \\
& \quad 3curF \leftarrow curF + 1 \\
& \quad 4\text{for all} \ J \in \text{Children}(I) \ \text{do} \\
& \quad 5\ \text{Build}(J, I.lr) \\
& \quad 6I.g \leftarrow curF - 1 \\
\end{align*}
\]
The tree is traversed in the pre-order sequence by calling recursively the Build procedure. \( parentLr \) is the \( lr \) value of the parent node. \( curF \) is the \( f \) value of current node respectively. Since each node is scanned maximum once, the complexity of the algorithm is \( O(n) \). Here \( n \) is the number of nodes of the tree.

4.4.2 Link Graph Construction

After building the tree structure, we can build the link graph in linear time of the number of reference edges. Suppose \( V_L = \{n_1, n_2, ..., n_t\} \). We assume that

\[
n_1.f < n_2.f < \ldots < n_t.f.
\]

It means the reference nodes are sorted by \( f \) parameter \(^1\).

Suppose \( N := \{(u,v) | u \rightarrow v\} \). The definition of the link graph implies that the link graph is build if and only if \( N \) is determined. More over the complexity of building the link graph equals to the complexity to determine \( N \). The following corollary will help us to determine the set \( N \).

**Corollary 4.2.** \( n_b \rightarrow n_c \) if and only if \( b = Max(A) \) in which

\[
A = \{a \in \{1, 2, \ldots, t\} | [n_c.f, n_c.g] \subseteq [n_a.f, n_a.g]\}
\]

**Proof.** Since \((f, g)\) is an AD system, for each \( a \in A \) \( n_a \) is an ancestor of \( n_c \).

Suppose \( n_a \) is an ancestor of \( n_b \). Since reference nodes are sorted by \( f \) values, we have: \( a < b \). Therefore, \( n_b \rightarrow n_c \) if and only if \( b = Max(A) \).

The algorithm determining \( N \) the set of pairs of \((a, b)\) in which \( n_a \rightarrow n_b \) is shown as follows:

**Closest-Reference-Ancestor(\(V_f\))**

begin
1 \( N \leftarrow \emptyset \)
2 \( S \leftarrow \emptyset \)
3 for \( i = t \) down to 1 do
4 begin
5 while \( S \neq \emptyset \wedge (n_{top(S)}.f \leq n_i.g) \) do
6 begin
7 \( N \leftarrow N \cup (n_i, n_{top(S)}) \)
8 \( S.Pop() \)
9 end

---

\(^1\)In the tree structure index constructor algorithm the nodes are automatically sorted by \( f \) parameter
At a point of time, each node is in one of following three states: (a) is not scanned; (b) is scanned and its index is an element of stack $S$; (c) is scanned and its index is removed from stack $S$.

**Corollary 4.3.** In the Closest-Reference-Ancestor algorithm the condition:

$$n_{\text{top}(S)}.f \leq n_i.g$$

is equivalent to

$$n_i.f < n_{\text{top}(S)}.f \leq n_{\text{top}(S)}.g \leq n_i.g$$

and the condition:

$$n_{\text{top}(S)}.f > n_i.g$$

is equivalent to

$$n_i.f \leq n_i.g < n_{\text{top}(S)}.f \leq n_{\text{top}(S)}.g$$

**Proof.** Since the nodes are scanned by the decrement of $f$ parameter so

$$n_i.f \leq n_{\text{top}(S)}.f$$

Since $(f, g)$ is an AD system, $[n_i.f, n_i.g]$ and $[n_{\text{top}(S)}.f, n_{\text{top}(S)}.g]$ are disjoint, or one of them is a subset of the other. Therefore,

$$n_i.f < n_{\text{top}(S)}.f \leq n_{\text{top}(S)}.g \leq n_i.g$$

or

$$n_i.f \leq n_i.g < n_{\text{top}(S)}.f \leq n_{\text{top}(S)}.g$$

Hence, the corollary is true.

**Corollary 4.4.** Suppose $v$ is in state (c) at the beginning of scanning some node. Then, there exists $u$ in state (b) such that $u$ is an ancestor of $v$.

**Proof.** Suppose $v$ is removed at line 8 when $u$ is scanned. Clearly, we have $v.f \leq u.g$ (the condition at line 5). Corollary 4.3 implies that $[v.f, v.g] \subseteq [u.f, u.g]$. It implies that $u$ is an ancestor of $v$. Hence, if a node is removed from $S$, an its ancestor is push into this stack. Therefore, the corollary is true.
Corollary 4.5. Suppose the elements of stack $S$ ordered by the increment of the time of pushing-in-stack are:

$$< a_1, a_2, \ldots, a_l >$$

Then, we have:

$$n_{a_l}.f \leq n_{a_l}.g < n_{a_{l-1}}.f \leq n_{a_{l-1}}.g < \ldots < n_{a_1}.f \leq n_{a_1}.g$$

Proof. Clearly, at the initial time when $S$ is empty, (2) is true. Moreover, the remove operations do not have any affect on the accuracy of the corollary.

Suppose at the beginning of scanning $n_i$

$$S = < a_1, a_2, \ldots, a_l >$$

and at the end of of scanning $n_i$

$$S = < a_1, a_2, \ldots, a_l, i >$$

in which $t \leq l; n_{a_{t+1}}, n_{a_{t+2}}, \ldots, n_{a_l}$ are removed out while $n_i$ is scanned; and

$$n_{a_l}.f \leq n_{a_l}.g < n_{a_{l-1}}.f \leq n_{a_{l-1}}.g < \ldots < n_{a_1}.f \leq n_{a_1}.g$$

Since $t$ is not removed from the stack, we have:

$$n_{a_t}.f > n_i.g.$$ 

Corollary 4.3 implies that:

$$n_i.f \leq n_i.g < n_{a_t}.f \leq n_{a_t}.g$$

Therefore, after scanning $n_i$, we have:

$$n_i.f \leq n_i.g < n_{a_t}.f \leq n_{a_t}.g < \ldots < n_{a_i}.f \leq n_{a_i}.g.$$ 

Hence, the corollary is true.

Corollary 4.6. Before scanning each node, for each node $u$ is in state (b) there does not exist $v$ in state (b) or (c) such that $v$ is an ancestor of $u$.

Proof. Suppose $u$ is an arbitrary node in state (b).

Suppose $u'$ is arbitrary nodes which are is in state (b). Applying Corollary 4.5, since $[u.f, u.g]$ and $[u'.f, u'.g]$ are disjoint, $u'$ is not an ancestor of $u$.

Suppose $v$ is an arbitrary nodes in state (c) and $v$ is an ancestor of $u$. According to Corollary 4.4, there exists $u'$ which is in state (b) such that $u'$ is an ancestor of $v$. Clearly, $u'$ is also an ancestor of $u$. It can not be true since $[u.f, u.g]$ and $[u'.f, u'.g]$ are disjoint. Therefore, there does not exist any node in state (c) which is an ancestor of $u$. 

\[\square\]
Proposition 4.7. The Closest-Reference-Ancestor algorithm is correct and stops after finite steps. The complexity of the algorithm is $O(k)$ where $k$ is the number of reference edges.

Proof. We show that if the index of $v$ is removed from the stack while $u$ is scanned then $u$ is the nearest reference ancestor of $v$.

At the beginning of scanning $u$, $v$ is in state (b). According to Corollary 4.6, there does not exist any ancestor of $v$ which is in state (b) and state (c). Therefore, $u$ is the ancestor of $v$ of which the $f$ value is highest. It implies that $u$ is the nearest reference ancestor of $v$.

At the end of the algorithm when all nodes are scanned, there is no nodes in state (a) and each node in state (c) has only one nearest reference ancestor which has been found by the algorithm. Furthermore, there does not exist two node $u$, $v$ in state (b) such that $u$ is an ancestor of $v$. Hence, the algorithm returns exactly $N$ set.

Each node is scanned, pushed into the stack and removed from the stack maximum once. Furthermore, we have the number of reference nodes is bound by $2k$. Hence, the complexity of the algorithm is $O(k)$.

4.5 Summary

In this chapter, we have introduced and studied on efficient primary query processing algorithms over the near tree XML data sets. Our ideas are based on the two facts: the processing costs of primary queries over the XML trees with the tree structure indexes are very low, and the number of reference edges is very small comparing with the number of edges. Two algorithms for the problem have been represented.

The principle of the first algorithm is the decomposition of the data graph into two components: the basic tree and the link graph. The basic tree is indexed by the tree structured indexes, with which we can compute the basic path connecting two arbitrary nodes in constant time. The link graph simulates all paths connecting between the reference nodes. The result of a primary query is determined by the processing over the basic tree and the link graph. Since the size of the basic tree is very small comparing with the data graph, the algorithm is more efficient than the naive graph traversal algorithm.

The second algorithm is the advanced version of the first one with the using of the structural indexes. We introduced the strong 1-index, which is a stable structural index. Since the basic tree structure is conserved, the strong 1-index is decomposed into the basic tree and the link graph. The new algorithm combines and takes the advantages of the tree structured- and the structural indexes. The experiments shown in Appendix B
show that the second algorithm overcomes the other algorithms. The average cost of the second algorithm is about 13% of the naive algorithm, and 36% of the first algorithm.

We have shown how to use the algorithms for constructing the tree structure index and the link graph. The complexity of the algorithm constructing the tree index structure is a linear function of the number of nodes. The complexity of the algorithm constructing the link graph is a linear function of the number of reference edges.
Chapter 5

Efficient Processing Of Regular Queries Over Fragmented XML Tree

One of the most motivated factors in the causation of the environment described in this chapter is *shared-nothing parallel database system* [8, 23, 24] (SNPDS for short). Figure 5.1 illustrates the model of SNPDS. A SNPDS consists of machines which are connected to each other by high-speed link. Each machine (or *site*) has a non-shared memory, a non-shared disk and a non-shared processor. The shared-nothing parallel database systems provide incremental and practically unlimited growth. Furthermore, in these systems, the failure is local. It means that if one site fails, the others still stay up. Nowadays, RAM and disk volumes grow quickly, the prices of processors become cheaper, and the speeds of local networks become higher. Hence, shared-nothing parallel database systems are being used in various areas of applications recently.

Figure 5.1: The model of share-nothing parallel database systems.
In this chapter, the considered XML data sets are very large trees which are fragmented over SNPDS. The fragmented XML tree is considered as the union of subtrees which are called fragments. Each site stores a set of fragments. The sets of fragments on each site are disjoint. The master site is chosen from the sites, and the remaining are slaver sites. Regular queries from clients are sent to the master site. The master controls the system to process the queries and send the results back to the clients.

There are two efficiency factors of query processing algorithms in SNPDS: (1) the waiting time for the result, and (2) the total query processing and communication cost over all machines of the system. There are two approaches for this problem: stream processing representing by the tree traversal algorithm \[4, 22\] and partial processing representing by the partial parallel processing algorithm \[4, 22\]. The tree traversal algorithm is quite similar to the naive graph traversal algorithm introduced in chapter 2. This algorithm does not use parallelism. Thus, the cost of communication and the waiting time for the result are very high. The partial parallel processing algorithm is based on the partial evaluation. Since parallelism is used, the cost of communication and the waiting time are reduced. However, since all operations are executed, the query processing cost is high.

A new efficient regular query processing algorithm in SNPDS is introduced and studied. The algorithm is also based on the partial evaluation. We define two types of unnecessary operations in the partial evaluation. The unnecessary operations type 1 are determined by the tree index. The unnecessary type 2 are restrained by the structural indexes. The sizes of the tree index and the structural indexes are chosen to fit the system. The sizes of these indexes can be considered as constants. Thus, the costs of processing over these indexes can be ignored. In this case, our algorithm overcomes the two above algorithms in both the waiting time and the total query processing and communication cost.

The rest of this chapter is organized as follows. Section 5.1 is the preliminary. The definition of the fragmented XML trees and the basic processing operation over fragment are introduced. Section 5.2 is about stream processing and partial processing approaches. We study on the two algorithms: tree traversal algorithm representing the stream processing approach and partial parallel processing algorithm representing the partial processing approach. In section 5.3, we introduce and study our algorithm. We define two types of unnecessary operations and describe the techniques to restrain them using tree-
structural indexes. Section 5.4 concludes the paper.

5.1 Preliminaries

5.1.1 Fragmented Tree

Definition 5.1. (Fragmented tree) A fragmented XML tree $T$ is denoted by $T = (F, S, E_l)$ in which:

i. $F$ is the set of fragments. Each fragment $F \in F$ is a subtree of $T$. They are disjoint and their nodes cover $T$. It means $\{V(F)\}_{F \in F}$ is a partition where $V(F)$ is the set of nodes of fragment $F$.

ii. $E_l$ is the set of link edges. Each link edge $(u, v) \in E_l$ connects from fragment $F$ to $F'$ ($F \neq F'$) if $v$ is the root of $F_j$, $u$ is a node of $F_i$ ($u \in V(F_i)$), and $(u, v)$ is an edge of $T$.

iii. $S$ is the set of sites. For each site $S \in S$, we have:

\[ S \subseteq F \land S \neq \emptyset. \]

Furthermore, the sites are disjoints:

\[ \forall S, S' \in S, S \neq S' : S \cap S' = \emptyset, \]

and they cover $F$:

\[ \bigcup_{S \in S} S = F. \]

iv. For each link edge connecting from fragment $F$ to $F'$, $F$ and $F'$ must be not in the same site.

For each fragment $F$, $\text{root}(F)$ denotes the root of fragment $F$. The fragment containing the root of the tree $T$, is called the root fragment denoted by $F_{\text{root}}$. In Figure 5.2, it is $F_0$. Given two fragments $F$ and $F'$, we say that $F$ is a child-fragment of $F'$ and $F'$ is the parent-fragment of $F$ if there exists a link edge from $F$ to $F'$. In Figure 5.2, fragment $F_5$ is a child-fragment of $F_2$ and $F_0$ is a parent-fragment of $F_1$. The link edges are: $(2, 5)$, $(7, 14)$, $(17, 24)$, $(1, 4)$ and $(10, 16)$. The (iv) condition in the definition avoids us the trivial case when a pair of "parent-children" fragments are stored in a same site.
5.1.2 Processing Over Fragment

Fragment is considered as the basic unit of the fragmentation, and the operation processing fragment beginning with some state of the query graph is considered as the basic operation of query processing. The basic operation processing fragment $F$ with state $q$ of query graph of regular expression $R$ is generalized as follows:

\[
\text{FRAGMENT-PROCESS}(F,q) \rightarrow \text{SCAN}(u,q)
\]

begin
1. $\text{FragResult} \leftarrow \emptyset$
2. $\text{PRE-PROCESS}(F,q)$
3. $\text{SCAN}($root$(F),q)$
4. $\text{POST-PROCESS}(F,q)$
5. return $\text{FragResult}$
end

The structure of procedure FRAGMENT-PROCESS is quite similar to the structure of procedure NAIVE-GRAF-TRAVERSAL introduced in chapter 2. Fragment $F$ is considered as a “small” data graph. Since $F$ is a tree each pair $(u, q)$ is visited maximum once\(^1\). Hence, the Visited, which is the list of visited pair of nodes $(u, q)$, is not necessary in procedure FRAGMENT-PROCESS. Three virtual procedures, WORKING-LINK-NODE, PRE-PROCESS and POST-PROCESS are designed for distributed environment. Concretely,

- WORKING-LINK-NODE procedure: Working with link edges.

---

\(^1\)See more detail in the proof of proposition 5
• **Pre-Process** procedure: Doing the works before the fragment processing.

• **Post-Process** procedure: Doing the works after the fragment processing.

The main differences between the query processing algorithms over distributed environment are presented by the implementation of these virtual procedures.

### 5.2 Stream- And Partial Processing Approaches

We study on two most popular approaches for processing regular queries in distributed environment: stream processing and partial processing. The tree traversal algorithm represented for stream processing algorithm approach is mentioned in subsection 5.2.1. The partial parallel processing algorithm represented for partial processing approach is investigated on in subsection 5.2.2.

#### 5.2.1 Tree Traversal Algorithm

In the stream processing approach, the fragmented tree is considered as the only one unified unit. The tree is processed fragment by fragment like a stream data. Concretely, suppose $e = (u, v)$ is a link edge connecting from fragment $F$ to fragment $F'$. Suppose $e$ is processed in the operation $(F, q)$ (line 4 in procedure Scan), and $p$ is the current state. The tree traversal algorithm will works as follows:

1. Operation $(F, q)$ is stopped and operation $(F', p)$ is setup.

2. Operation $(F', p)$ is executed.

3. When $(F', p)$ is finished, its result will be sent to the operation $(F, q)$.

4. Operation $(F', p)$ is resumed and continues processing at the next command where it was stopped.

Formally, the **Post-Process** and the **Working-Link-Edge** procedures are implemented as follows:

**Post-Process**$(F, q)$  
begin  
1. if $F \neq F_{root}$ then  

**Working-Link-Edge**$((u, v), p)$  
begin  
1. Let $F'$ be the remote fragment containing $v$
2. **Send-Result**(Result)  2. **Fragment-Process**(F\(^{\prime}\), p) (Calling from remote)

**end**

3. subResult ← **Receive-Result**(F\(^{\prime}\))
4. **FragResult** ← subResult ∪ **FragResult**

**end**

The **Send-Result** procedure sends the result to the parent-fragment. The **Receive-Result**(F\(^{\prime}\)) procedure receives the result from the operation over child fragment F\(^{\prime}\). Each execution of the **Receive-Result** procedure synchronizes with some execution of the **Send-Result** procedure. Finally, the result of the query on tree \(T\) is the result of the fragment processing operation (\(F_{\text{root}}, q_{\text{root}}\)). It is determined by calling the **Spider** procedure.

**Spider**(Q)
begin
1. return **Fragment-Process**(\(F_{\text{root}}, q_{\text{root}}\))
end

Let’s consider the example of fragmented tree and query graph shown in Figure 5.2. The sequence of events of the tree traversal algorithm in this case is:

1. (\(F_0, q_0\)) starts.
2. Link edge (2,5) is processed with current state \(q_0\). (\(F_0, q_0\)) stops. (\(F_1, q_0\)) starts.
3. (\(F_1, q_0\)) finishes. The result is empty set.
4. (\(F_0, q_0\)) is resumed.
5. Link edge (2,5) is processed with current state \(q_2\). (\(F_0, q_0\)) stops. (\(F_1, q_2\)) starts.
6. (\(F_1, q_2\)) finishes. The result is \{5\}, and it is sent to operation (\(F_0, q_0\)).
7. (\(F_0, q_0\)) is resumed.
8. ...

The tree traversal algorithm can be considered as a sequence of executions of the **Fragment-Process** procedure. There is no parallelism. Each site and fragment can be visited in many times. Moreover, the number of the communication among sites depends on the query. Therefore, the waiting time is very high, and it does not take the advantages of the share-nothing parallel database systems.

Using the tree traversal algorithm, we define the reachable operations as follows:

**Definition 5.2.** (Reachable operation) An operation \((F, q)\) is reachable if it is executed by the tree traversal algorithm.

The reachable operations in the example are: \((F_0, q_0), (F_1, q_0), (F_1, q_2), (F_2, q_0), (F_5, q_0), (F_5, q_1), (F_3, q_0), (F_3, q_1), (F_4, q_0), (F_4, q_3)\).
5.2.2 Partial Parallel Processing Algorithm

The main behavior of the partial processing approach is that each operation is executed partially. While a operation is processed there is no communication with other operations. If a link edge is processed, we just write down the relationship between the current operation and the corresponding operation which is executed over the child fragment. Basing on these information, we determine the reachable operations. The Pre-Process and the Working-Link-Edge procedures are implemented as follows:

\[
\text{Pre-Process}(F,q) \quad \text{Working-Link-Edge}((u,v),p)
\]

\[
\begin{array}{ll}
\text{begin} & \text{begin} \\
1. bF \leftarrow F & 1. \text{Let } F' \text{ be the remote fragment containing } v \\
2. bQ \leftarrow q & 2. Map \leftarrow Map \cup ((bF,bQ),(F',p)) \\
\text{end} & \text{end}
\end{array}
\]

The fact "if \((bF,bQ)\) is processed, \((F,q)\) will be processed." is coded by \(((bF,bQ),(F,q))\). Here \((bF,bQ)\) is current operation, and \((F,q)\) is the operation which is executed over child-fragment \(F\). \(bF\) and \(bQ\) variables are initialized in the Pre-Process procedure. \(Map\) is the global variable storing all relationships of operations we find out when processing all operations over this site. The operations over sites are executed partially and in parallel.

At each site, all possible operations are executed firstly. Then, the map of operations is sent to the master site. Basing on all the maps of all sites, the master determines the reachable operations by applying inductively two following rules:

1. \((F_{\text{root}},q_{\text{root}})\) is reachable.

2. Suppose \(((F,q),(F',q'))\) is a fact. Then, if \((F,q)\) is reachable, \((F',q')\) is reachable, too.

Each site waits until the master site sends the set of reachable operations over it. Finally, all sites just send the reachable results to the master site and the union of these results is the result of the query.

Formally, Master\((Q)\) procedure runs in the master site to control the sites processing query graph \(Q\). The partial result of query graph \(Q\) is computed in site \(S\) by using the Site-Process\((S,Q)\) procedure. The reachable operations are determined by the Compute-Reachable-Results procedure on the master site. In the Master procedure, \(Map[S]\) is the map of the operations we receive from site \(S\), and \(RO[S]\) is the set of reachable operations of site \(S\).
Master($Q$)
begin
1. for each site $S$ do in parallel
2. Site-Process($S,Q$) (Calling from remote)
3. Map[$S$] ← Receive-Map($S$)
4. joint: waiting until receiving all map result from every site
5. Compute-Reachable-Operations()
6. for each site $S$ do in parallel
7. Send-Reachable-Operations($S,RO[S]$)
8. subResult[$S$] ← Receive-Result($S$)
9. joint: waiting until receiving all result from every site
10. Result ← $\emptyset$
11. for each site $S$ do
12. Result ← Result $\cup$ subResult[$S$]
end

Site-Process($S,Q$)
begin
1. Map ← $\emptyset$
2. for each fragment $F$ of $S$ do
3. for each state $q$ of $Q$ do
4. result[$F,q$] ← Fragment-Process($F,q$)
5. Send-Map(Map)
6. RO ← Receive-Reachable-Operations()
7. Result ← $\emptyset$
8. for each $(F,q) \in RO$ do
9. Result ← Result $\cup$ result[$F,q$]
10. Send-Result(Result)
end

The sequence of the activities ordered by time:

1. Activating the query process over each site (line 1-2 of the Master procedure).
2. Computing the partial result at each site (line 1-4 of the Site-Process procedure).
3. Synchronization: Sending the map of the result from some site to the master site (line 5 in the Site-Process procedure, line 3 in the Master procedure).
4. The master waits until receiving all map results (line 4 in the Master procedure).
5. Determining reachable operations (line 5 in the Master procedure).
6. Synchronization: Sending the reachable results to each site (line 6 in the Site-Process procedure, line 7 in the Master procedure).
7. Each site determines the site result (line 7-9 in the Site-Process procedure).

8. Synchronization: Sending the result from some site to the master site (line 10 in the Site-Process procedure, line 8 in the Master procedure).

9. The master waits until receiving all site results (line 9 in the Master procedure).

10. Determining the final result (line 10-12 in the Master procedure).

Let’s see the example shown in Figure 5.2. In the case the operation \((F_2, q_3)\) is executed, there is no communication with the operations over \(F_0\) or \(F_5\). The result of this operation is \(\{17\}\). We execute all operations in site \(S_0, S_1\) in parallel. In site \(S_1\), we execute 12 operations:

\[(F_1, q_0), \ldots, (F_1, q_3), (F_2, q_0), \ldots, (F_2, q_3), (F_3, q_0), \ldots, (F_3, q_3)\]

From operations over \(F_2\) and we will find out following facts of relationship of the operations:

- if \((F_2, q_0)\) is processed, \((F_5, q_0)\) and \((F_5, q_1)\) will be processed
- if \((F_2, q_1)\) is processed, \((F_5, q_2)\) will be processed
- if \((F_2, q_2)\) is processed, \((F_5, q_2)\) and \((F_5, q_2)\) will be processed

Suppose \(S_0\) is the master site. Basing all map of operations from the sites \(S_0\) will determined the reachable operations. Finally, each site only sends the sub-results of reachable operations, which has been computed in previous phases, to the master site. Site \(S_1\) has sent the results of following operations: \((F_1, q_0), (F_1, q_2), (F_2, q_0), (F_3, q_0), (F_3, q_1)\).

The waiting time is reduced by the parallelism. Data of each site and fragment is processed only once. Moreover, the number of the communication between sites is constant, and the size of communicated data depends only on the size of the query and the result [22]. We can reduce the waiting time by expanding the system with more machines (sites). However, the total processing cost is high since all operations are executed.

### 5.3 Efficient Processing Regular Queries Using Tree And Structural Indexes

We define two types of unnecessary fragment processing operations in partial processing approach.
Definition 5.3. (Unnecessary fragment processing operations) \((F,q)\) is unnecessary type 1 if it is unreachable. \((F,q)\) is unnecessary type 2 if this operation returns empty set.

In the example shown in Figure 5.2, \((F_2,q_2)\) is an unnecessary operation type 1 since from all operations over \(F_2\), \((F_2,q_0)\) is the only operation which is reachable from \((F_0,q_0)\). However, it is not unnecessary type 2 since if \((F_2,q_2)\) is processed, node 18 matches the query. In another case, \((F_4,q_0)\) is an unnecessary operation type 2 but not an unnecessary operation type 1. Obviously, we do not need to carry out an operation if it is unnecessary type 1 or 2.

Our efficient algorithm is also based on partial evaluation. The unnecessary operations are restrained by preprocessing over the tree- and the structural indexes on the master site. In subsection 5.3.1, we introduce and study on the tree index which is used for rejecting unnecessary operations type 1. In subsection 5.3.2, we mention the technics for restraining the unnecessary operations type 2 by using the structural indexes. Subsection 5.3.3 will describes our algorithm.

5.3.1 Tree Indexes

Definition 5.4. (Tree index) Tree \(T_I = (V_I, E_I, label_I, label'_I)\) is the tree index of the XML tree \(T = (\mathcal{F}, \mathcal{S}, E_l)\) in which:

i. \(V_I = \mathcal{F}\) is the set of index nodes, and \(label : V_I \mapsto \Sigma\) is a labeling function. For each \(F \in \mathcal{F}\) \(label_I(F)\) is defined as \(label(\text{root}(F))\).

ii. \(E_I \subseteq V_I \times V_I\) is the set of index edges in which \((F,F') \in E_I\) if \(F\) is parent fragment of \(F'\). \(label'_I : E_I \mapsto \Sigma^*\) is a labeling function. Each edge \(e = (F,F')\) is labeled by \(\alpha = label'_I(e)\) in which \(\alpha\) is the label value of the path connecting from \(\text{root}(F)\) to \(\text{root}(F')\) rejecting the roots of \(F\) and \(F'\).

The tree index stores all information about the paths connecting between the roots.
of the fragments. The index tree of the fragmented tree $T$ in Figure 5.2 is depicted in Figure 5.3(a). In this tree, $(F_0, F_2)$ is labeled by $a.c$ which is the label value of the path $<3,7>$; $(F_0, F_3)$ is labeled by $\varepsilon$ as the root of $F_0$ is the parent of the root of $F_3$. The size of the index tree can be considered as constant as the number of fragments is considered as constant.

The reachable operations which are not unnecessary type 1 are determined by the Process-Index-I procedure as follows:

**Process-Index-I($T_I, Q$)**  
**begin**  
1. $RO \leftarrow \emptyset$  
2. $Stack \leftarrow \emptyset$  
3. $Stack.push((F_{root}, \text{Root}(Q)))$  
4. **while** $Stack \neq \emptyset$ **do**  
5. $(F, q) \leftarrow Stack.pop()$  
6. $RO \leftarrow RO \cup (F, q)$  
7. **for each** index edge $(F, F')$  
8. **for each** $q' \in \text{Matching-II}((F, F'), q)$  
9. $Stack.push((F', q'))$  
10. **return** $RO$  
**end**

The structure of Process-Index-I procedure is similar to structure of the Compute-Reachable-Results procedure in the partial parallel processing algorithm. The difference between two procedures is the way expanding new reachable operations. We use the Matching-II procedure (at line 8) in the first procedure and $\text{Map}[S]$ (at line 9) and in the second. Both of them give the same information that "Which operations over the child-fragment are reachable if some operation over parent-fragment is reachable." The correctness of the Matching-II can be proven easily as it is considered as a sequence of applying the $E_R$ function. Hence, the Process-Index-I procedure will return the set of reachable operations. The processing cost over tree index is considered as constant if the size of the tree index is considered as constant.

### 5.3.2 Structural Indexes

We restrain redundant operations type 2 by processing the query graph over structural indexes of the fragments. The safeness of structural indexes is applied in this case. Let us remind the reader about the sufficient safeness condition of structural indexes introduced
in chapter 3. A structural index is safe with any SAPE queries if each data node path is simulated by some index node path.

Generally, suppose $I_G$ is a structural index of data graph $G$. We define the result of operation $(I_G, q)$, $R_I$, is the set of index nodes $U$ in which $(U_{\text{root}}, U)$ matches $(q, f)$. Here the root of data graph, $r$, is simulated by $U_{\text{root}}$, and $f$ is a final state. Then, we have:

**Proposition 5.1.** If each data node path is simulated by some index node path, we have:

$$\forall u \in R_G, \exists U \in R_I : u \in U.$$ 

**Proof.** Suppose $u$ is an element of $R_G$. There exists a final set $f$ such that $(r, u)$ matches $(q, f)$. Let $p = u_1u_2 \ldots u_l$ be the data node path connecting from $r$ to $u$, and $p_q = q_0q_1 \ldots q_l$ be the query node path connecting from $r$ to $f$ in which $p$ matches $p_q$. Let $p_I = U_0U_1 \ldots U_l$ is the index node path which simulates $p$. Since $p$ matches $p_q$, we have:

$$\forall i = 1, 2, \ldots, l : q_i \in E_R(q_{i-1}, \text{label}(u_{i-1})).$$

Since $\text{label}(u_i) \in \text{label}(U_i)$, we have:

$$\forall i = 1, 2, \ldots, l : q_i \in E_R(q_{i-1}, \text{label}(U_{i-1})).$$

Hence, $(U_0, U_l)$ matches $(q, f)$. Because $U_0$ simulates $r$, we have: $U_l \in R_G$. Therefore, the proposition is true.

Suppose $I_F$ is a structural index of fragment $F$ in which each data node path is simulated by some index node path. Thus, $I_F$ is safe with any regular query. We can check if $(F, q)$ is an unnecessary operation type 2 by preprocessing over $I_F$. If the operation $(I_F, q)$ returns no result, the operation $(F, q)$ also returns no results.

Clearly, we can use any ”safe” structural index for restraining the unnecessary operation type 2. However, the preprocessing cost should be minimized. Thus, the size of the structural indexes of the fragments should be very small. For this purpose, we would like to introduce the DL-tree-indexes for simulating fragments. These indexes are safe with any regular queries, theirs size are very small, and they are quite adaptive.

We define special label $\ast'$ (*$ \notin \Sigma$) for matching any label of $\Sigma$. If the label of an index node $I$ is $\ast'$, we can jump from $(I, q)$ to any $(J, p)$ in which $(I, J)$ is an index edge and there exists a transition from state $q$ to state $p$.

**Definition 5.5.** (DL-tree-index) Let $F$ be the fragment, $h$ be the depth of $F$ and $k$ be an integer $1 \leq k \leq h$. The $k$-depth DL-tree-index of $F$, $I_k$, is a structural index in which the partition of nodes is as follows:
i. For each $0 \leq l \leq k$, we group the data nodes in the same depth $l$ in the same index node $I_l$.

ii. In the case $k < h$, we group the data nodes whose depth is more than $k$ in the same index node and the index node $I_{k+1}$ labeled by 'ψ'.

iii. There exists an index edge from $U$ to $U'$ if there exists an edge $(u, v)$ in which $u \in U$ and $u' \in U'$.

Clearly, the DL-tree-indexes are safe with any regular queries as each data node path is simulated by some index node path. The DL-tree-indexes of fragment $F_4$ of tree $T$ in Fig. 5.2(b) is shown in Fig. 5.3(b). The root of $\mathcal{I}_k$ is $I_0$. The $\mathcal{I}_k$ has $k$ index nodes in the case $k = h$ otherwise $k + 1$ index nodes. We can choose $k$ value as bigger as the size of $\mathcal{I}_k$ can be considered as constant.

In chapter 3, we have shown that the DL-indexes are quite dynamic and flexible. We can use a DL-tree-index as the initial index graph, and then refine the index so that the size fits to the system or the index graph supports a set of frequently queries.

### 5.3.3 Processing Queries With Tree- And Structural Indexes

The tree index $T_I$ of the system is stored on the master site. Moreover, for each fragment $F$, $\text{Index}(F)$ which is a $k$-depth DL-tree-index of $F$ is also stored on the master site. The $\text{Eff-Master}(Q)$ procedure is run in the master site to control the sites processing query graph $Q$. The partial result of query graph $Q$ is computed in site $S$ by using the $\text{Eff-Site-Process}(S,Q)$ procedure. The $\text{Process-Index-II}(\text{Index}(F), q)$ procedure will execute the operation $(\text{Index}(F), q)$. If there is some index node matching the query, $\text{Process-Index-II}(\text{Index}(F), q)$ will return true value immediately; otherwise, it will return false value.

\begin{verbatim}
\text{Eff-Master}(Q) begin
1. States ← Process-Tree-I($T_I, Q$)
2. for each site $S$ do
3. $RO[S] ← \emptyset$
4. for each $(F, q) \in \text{States}$ do
5. if Process-Index-II($\text{Index}(F), q$) then
6. Let $S$ be the site containing $F$
7. $RO[S] ← RO[S] \cup (F, q)$
8. for each site $S$ do in parallel
9. $\text{Eff-Site-Process}(S,Q)$ (Calling from remote)
10. $\text{Send-Required-Operations}(S,RO[S])$
end
\end{verbatim}
11. \( \text{subResult}[S] \leftarrow \text{RECEIVE-SITE-RESULT}(S) \)
12. **joint:** waiting until receiving all result from every site
13. \( \text{Result} \leftarrow \emptyset \)
14. **for each** site \( S \) **do**
15. \( \text{Result} \leftarrow \text{Result} \cup \text{subResult}[S] \)
**end**

The sequence of the activities of the query process ordered by time is:

1. **Rejecting the redundant operations at master site:** (line 1-7 of the `Master` procedure).
2. **Activating the query process at each site** (line 9 in the `Eff-Master` procedure)
3. **Synchronization: Sending the required operations to each site** (line 10 in the `Eff-Master` procedure, line 1 in the `Eff-Site-Process` procedure).
4. **Computing the required operations at each site.** (Line 3-4 in the `Eff-Site-Process` procedure)
5. **Synchronization: Sending the site result from some site to the master site** (line 5 in the `Eff-Site-Process` procedure, line 11 in the `Eff-Master` procedure).
6. **The master waits until receiving all site results** (line 12 in the `Eff-Master` procedure).
7. **Determining the result** (line 13-15 in the `Eff-Master` procedure).

Obviously, the number and the cost of communication in our algorithm are smaller and lower than two algorithms introduced in subsection 5.3. The total cost of query processing in each site is also smaller as the number of processing operations is fewer. If the number of fragments is considered constant, we can ignore the cost of processing over tree- and structural indexes. In this case, our algorithm overcomes two above algorithms according both the waiting time and the total query processing and communication cost criteria.

5.4 **Summary**

We have given an overview of the processing of regular queries over fragmented trees in SNPDS. There are two approaches for this problem: stream processing and partial processing. There is no parallelism in the stream processing approach. The partial processing approach takes the advantages of the parallelism, but the processing cost is still high as all operations are executed.
An efficient regular query processing algorithm in SNPDS based on partial evaluation has been introduced. Two types of redundant operations are defined. The redundant operation type 1, which are not reachable, are determined by the tree structural. The redundant operations type 2, which have no matching node, are restrained by processing over structural indexes. The DL-tree-indexes are introduced and proposed for this purpose. The size of the tree index and the DL-tree indexes are considered as constants. The partial evaluation becomes more effective by restraining redundant operations. Our experimental study in Appendix C has verifies the effectiveness of our techniques.
Chapter 6

Summary Of The Dissertation

We have studied some applications of using structural indexes for processing regular queries over XML data sets in different contexts.

Chapter 2 gives us an overview of processing regular queries over the XML data sets. We have introduced the graph models of XML data sets and regular queries. There are two different versions for representing data graph: node-labeled data graph and edge-labeled data graph. Both are rooted graphs whose basic structure is a tree. Query graph is the transition graph of the automata computing the regular query.

The definition of the result of a query graph over a data graph bases on the concept about a data node path matches a query node path. We have introduced two naive algorithms for computing the result of a regular query over a data graph: the NAIVE-GRAPH-TRAVERSAL algorithm in the case of node-labeled data graphs and the NAIVE-JOINING-GRAPH algorithm in the case of edge-labeled data graphs. The NAIVE-GRAPH-TRAVERSAL algorithm traverses both of data graph and query graph together to find all data node paths matching some query node path. In the NAIVE-JOINING-GRAPH algorithm, the result is determined by computing the transitive closure of the state-data graph which is the result of joining the data graph and the query graph. Both of these two algorithms have the same complexity but we, by using the NAIVE-JOINING-GRAPH algorithm, can store, manage and process data graphs and query graphs in legacy database system.

In Chapter 2, we have also studied the equivalence and the decomposition of regular queries. Two regular queries are equivalent over a given data graph if their results over this data graph given by arbitrary input set and output set coincide. We have showed that
for each regular query, there exists a finite sub-query which is equivalent to this query on a given data graph. Furthermore, each query can be decomposed into the primary queries.

Chapter 3 gives us an overview of structural indexes, and studies on the mathematical background, SAPE queries and the dynamic labeling techniques. Our mathematical background is based on the split, refine and CRF operations of partitions. It is used to model, create and refine the structural indexes. SAPE is a general type of the near primary query and designed for supporting wild- or alternation cards. Classical indexes with static labeling do not well support SAPE queries. The dynamic labeling techniques based on our mathematical background not only make the size of an index graph smaller, but also support the SAPE queries better. Furthermore, they make the index graph more dynamic and adaptive.

Every static labeling structural index can be improved so that it supports dynamic labeling. In chapter 3, we have studied the DL-1-index improved from the 1-index and the DL-A*(k)-index improved from the A*(k)-index. The indexes are built around the properties of the stable structural index and the k-stable structural index. The sizes of the DL-1-indexes are bound by that of the 1-index, and the sizes of the DL-A(k)-indexes are bound by that of the A*(k)-index. With the results of our experiments shown in Appendix A we believe that the DL-A*(k)-index is one of the most efficient adaptive structural indexes.

In Chapter 4, we have introduced and studied on the efficient primary queries processing algorithms over the near tree XML data sets. In the case of XML data sets, the number of reference edges is very small comparing with that of the edges. The underlying principles of the algorithms are the decomposition of the data graph into the link graph and the basic tree, and the usage of the tree structured index of the basic tree. The tree structured index can compute the basic path connecting two arbitrary nodes in constant time. Hence, the costs of processing primary queries over the XML trees with the tree structure indexes are very low. The link graph simulates all paths connecting the reference nodes. The size of the link graph is very small comparing with the data graph since the number of reference edges is very small comparing with that of the edges. The result of a primary query over the data graph is determined by processing over the basic trees.
and the link graph.

The advanced version of the above algorithm based on the strong 1-index, which is a stable structural index, is also introduced. Since the strong 1-index conserves the basic tree, the algorithm works over the index graph. We have presented algorithms for constructing the tree structure index and the link graph. The complexity of the algorithm constructing the tree index structure is a linear function of the number of nodes while the complexity of the algorithm constructing the link graph is a linear function of the number of reference edges.

In Chapter 5, we have studied the processing of regular queries over very large XML trees which are fragmented over SNPDS. We have described two algorithms for this problem: partial processing algorithm and tree traversal algorithm. There is no parallelism in the tree traversal algorithm. The partial processing takes the advantages of parallelism. However, there are many unnecessary operations since all fragment operations are executed.

Our algorithm is also based on the partial evaluation but the unnecessary operations are restrained by tree index and structural indexes. We have defined two types of unnecessary fragment operations. The unnecessary fragment operations type 1 are unreachable operations. They are determined by preprocessing on the tree index. The tree index stores all information about the paths connecting among the roots of fragments. The number of nodes of the tree index is equal to the number of fragment. The unnecessary operations type 2 are those operations which return no result. They are restrained by preprocessing on the structural indexes of the fragments. Since the tree- and the structural indexes are small our algorithm overcomes the classical ones in both the waiting time and the total cost of query processing and communication.
Appendix A

Experiments For DL-Indexes

We examined the performances of the $DL-I$-index vs. the 1-index, the $DL-A^*(k)$-index vs. the $A(k)$-index on the same large data sets with the different finite sets of $SAPE$’s $^1$. The experiments were performed on Celeron R (2.4 G.hz), platform with MS-Windows XP and 512 MBytes of main memory. The Xerces Java SAX parser 1 [27] was used to parse XML data. We implemented the algorithms in C++. The data sets and the sets of frequently used $SAPE$’s were chosen as follows.

Data sets. We used two data sets: the $XMark$ data set (100 Mb) and the $TreeBank$ data set (82M). The $XMark$ data set containing the activities of an auction Web site is generated by using the Benchmark Data Generator [29]. The $TreeBank$ data set is the collection of English sentences, tagged with parts of speech [30]. The properties of the data sets are as in Table A.1.

Query loads. For the simplicity, we assumed that in our $SAPE$ queries $S_i$ is a wild card or unique label value ($S_i = \Sigma$ or $|S_i| = 1$). For each data set, we generated 4 query loads randomly. Each query load contained 100 $SAPE$ queries. The length of each query

---

$^1$This appendix is the experiment of our paper "Efficient Processing SAPE Queries Using The Dynamic Labelling Structural Indexes" [P3].

<table>
<thead>
<tr>
<th>Data set</th>
<th>Number of Nodes</th>
<th>Number of Edges</th>
<th>Number of Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMark</td>
<td>1,681.342</td>
<td>1,987.929</td>
<td>76</td>
</tr>
<tr>
<td>TreeBank</td>
<td>2,437.667</td>
<td>2,437.666</td>
<td>251</td>
</tr>
</tbody>
</table>

Table A.1: The properties of the data sets using in the experiments of DL-indexes.
(A) XMark: DL-1-index vs. 1-index (Index’s size)

(B) TreeBank: DL-1-index vs. 1-index (Index’s size)

(C) XMark: DL-1-index vs. 1-index (Cost of Eval.)

(D) TreeBank: DL-1-index vs. 1-index (Cost of Eval.)

(E) XMark: DL-A*(4)-index vs. A(4)-index (Index’s size)

(F) TreeBank: DL-A*(4)-index vs. A(4)-index (Index’s size)

(G) XMark: DL-A*(4)-index vs. A(4)-index (Cost of Eval.)

(H) TreeBank: DL-A*(4)-index vs. A(4)-index (Cost of Eval.)

Figure A.1: The results of the experiments
was less than 5. In the $i$-th query load the probability of $S_i$ is a wild card was equal to $(i - 1) \times 20\%$ (In the query load 1, there did not exist wild card, and in the query load 2 the probability of $S_i$ be wild card was 20\%, etc...).

The sizes of index graphs and the average of the evaluation cost of the queries in the query loads were focused in the performance of the DL-1-indexes vs. the 1-index, the DL-A*(k)-index vs. the A(k)-index. We measured the size of the index graph by the number of index nodes. Similarly to [6, 11, 12] we adopted the same main-memory cost metric for query evaluation. The cost was the number of the visited index nodes. The validation was not necessary because each query load was supported by the corresponding dynamic labeling index graphs.

A.1 DL-1-Index vs. 1-Index

The results of the DL-1-index vs. the 1-index experiments are shown in Figure A.1 (A, B, C and D). In each chart, we represent the results of 4 tests corresponding with 4 query loads. The black color corresponds to the 1-index, and the white color corresponds with to DL-1-index. Compared with the 1-index, the sizes of the DL-1-indexes are smaller and the average cost of the query evaluation of the DL-1-index was lower, since the 1-index is a refinement of the DL-1-indexes. However, the sizes of the DL-1-indexes increased very fast after each refinement and converge to the size of the 1-index since they must fulfill to be stable structural indexes. In test 3 the proportions of the index size the DL-1-index vs. the 1-index were 80, 70\% (XMark) and 83, 41\% (TreeBank). As a result the efficiency of the DL-1-indexes decreased quickly after each refinement. In test 3 the proportions of the average cost of the query evaluation the DL-1-index vs. the 1-index were 81, 55\% (XMark) and 84, 87\% (TreeBank).

A.2 DL-A*(k)-Index vs. A(k)-Index

Because the length of each query was less than 5, we chose $k = 4$. The DL-A*(k)-index were much more dynamic than the DL-1-indexes since the index nodes of DL-1-indexes must be stable. Although the ancestor bisimilarity layers are stored by the DL-A*(k)-indexes, but in our tests the sizes of these indexes were smaller than that of the A(k)-indexes. In test 3, the proportions of the index size between the DL-A*(k)-index vs. the
A(k)-index were 53, 58% (XMark) and 40, 87% (TreeBank). By using ancestor bisimilarity layers the cost of query evaluation on the DL-A*(k)-index was quite cheaper than on the A(k)-index. In test 3, the proportions of the average cost of the query evaluation the DL-A*(k)-index vs. the A(k)-index were 8, 46% (XMark) and 4, 97% (TreeBank). Because of the above results we believe that the DL-A*(k)-index is one of the most efficient adaptive structural indexes.
Appendix B

Experiments For NA-, NI-, NT- And TI Algorithms

We compared the performance among 4 algorithms: the naive algorithm (NA algorithm), the naive algorithm over the 1-index (NI algorithm), the improved algorithm (NT algorithm), and the improved algorithm for the strong 1-index (TI algorithm) on the same large data sets. The experiments were performed on Celeron R (2.4 G.hz), platform with MS-Windows XP and 512 MBytes of main memory. The Xerces Java SAX parser and the xmlgen the The Benchmark Data Generator were used to parse and to generate XML data. We have implemented the 4 algorithms in PL/SQL language and represented the data sets in ORACLE 9.i. The data sets and the query workload were chosen as follows.

Data sets. Using the Benchmark Data Generator, we generated three data sets $D_0$, $D_1$ and $D_2$ with the sizes 17MB, 30MB and 47 MB respectively. The properties of 3 data sets were as below:

For each data set we also built the 1-index and strong 1-index. The properties of the index graphs were shown in Table B.2:

Query Workload. We generated 144 primary queries above 3 data sets. The lengths of queries were from 8 to 12. For the simplicity, we assume that the input set contains only the root, $I = \{root\}$, and the output set is the whole set of nodes, $O = V$. We used the top down approach for the query evaluation.

\[\text{This appendix is the experiment of our paper "Combining Tree Structure Indexes With Structural Indexes" [P2]}\]
Table B.1: The properties of the data sets using in the experiments of processing algorithms over near tree XML data sets.

<table>
<thead>
<tr>
<th>Data set</th>
<th>No. of Nodes</th>
<th>No. of Edges</th>
<th>No. of ref. edges</th>
<th>No. of Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_0$</td>
<td>250732</td>
<td>296101</td>
<td>45370</td>
<td>76</td>
</tr>
<tr>
<td>$D_1$</td>
<td>421006</td>
<td>497491</td>
<td>76486</td>
<td>76</td>
</tr>
<tr>
<td>$D_2$</td>
<td>673241</td>
<td>796329</td>
<td>123089</td>
<td>76</td>
</tr>
</tbody>
</table>

Table B.2: The properties of the 1-index and the strong 1-index using in the experiments of processing algorithms over near tree XML data sets.

<table>
<thead>
<tr>
<th>Data set</th>
<th>1-index</th>
<th>strong 1-index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of Nodes</td>
<td>No. of Edges</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>$D_0$</td>
<td>113004</td>
<td>150353</td>
</tr>
<tr>
<td>$D_1$</td>
<td>189461</td>
<td>252451</td>
</tr>
<tr>
<td>$D_2$</td>
<td>302663</td>
<td>404432</td>
</tr>
</tbody>
</table>

B.3 Performance Result

Cost model. To our knowledge, there is no standard storage scheme and query cost model for graph structured data. Hence, we adopted the same main-memory cost metric similar to those used in [6,12]. The cost of the NI and NA algorithm was the number of visited edges in step 2, when we compute the transitive closure relation of the state data graph. The cost of the NT and TI algorithms with top-down approach was the sum of following numbers: (1) The number of the records in relation $R_1$ in step 2; (2) The number of the records in relation $R'_1$ (3) The number of the visited edges of the link graph in relation $R'_2$ using $R'_1$ as the first rule [P1] (4) The number of the records in relation $R'_3$ as we use $R'_2$ as semi join precondition [P1]. Notice that we did not count the data nodes in the extents of index nodes in the TI and NI algorithms, and the complexities in join operations were ignored.

With the above cost metric the average of the costs of the queries evaluation with 4 algorithms were shown in Table B.3.

The average cost of the TI algorithm was about 13% of the N algorithm’s, about 36%
Table B.3: The average of the costs of the queries evaluation with N-, NI-, NT- and TI algorithms.

<table>
<thead>
<tr>
<th>Data set</th>
<th>Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N</td>
</tr>
<tr>
<td>$D_0$</td>
<td>4780</td>
</tr>
<tr>
<td>$D_1$</td>
<td>8146</td>
</tr>
<tr>
<td>$D_2$</td>
<td>13088</td>
</tr>
</tbody>
</table>

of the NI algorithm’s and about 68% of the NT algorithm’s. The results demonstrated that the performance of the TI algorithm overcame the other algorithms.
Appendix C

Experiments For Different Regular Query Processing Algorithms Over Fragmented XML Tree

We examined the performance of our efficient algorithm using tree and structural indexes (EPP for short), the partial processing algorithm (PP for short) and the tree traversal algorithm (TP for short) in the waiting time and the total processing and communication criteria \(^1\). We implemented the algorithms in C++ programming language.

Data set and fragmentation. We used the XMark data set containing the activities of an auction Web site was generated by using the Benchmark Data Generator [29]. The size of our data set was about 500 Mb. The number of nodes was 10 267 360 and the number of labels was 77. We used 19 Linux machines which were connected by a local LAN. We used the program splitting the XML tree into 76 fragments. Each fragment had about 150 000 nodes. The fragments were chosen randomly to be stored in the sites.

Queries. We proposed 10 regular path queries representing for different conditions of the environment for our experiments \(^2\). They were:

1. Q1: //keyword.
4. Q4: //listitem//keyword
5. Q5: //people/*/profile/income

\(^1\)This appendix is the experiment of our paper “Efficient Processing Regular Queries In Shared-Nothing Parallel Database Systems Using Tree- And Structural Indexes” [P5]

\(^2\)These queries are written in XPath syntax
6. Q6: bold | emph
7. Q7: asia/bold | asia/emph
8. Q8: person/name
9. Q9: item/mailbox/mail
10. Q10: namerica/description/keyword

We processed these queries with our algorithms many times and measured the average values of the waiting time and the cost of processing and communication for each query and each algorithm. The results are shown in Table C.1.

<table>
<thead>
<tr>
<th>Queries</th>
<th>Waiting time(ms)</th>
<th>Pro. &amp; Com. time(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TP</td>
<td>PP</td>
</tr>
<tr>
<td>Q1</td>
<td>138662</td>
<td>4590</td>
</tr>
<tr>
<td>Q2</td>
<td>128900</td>
<td>1987</td>
</tr>
<tr>
<td>Q3</td>
<td>83516</td>
<td>2638</td>
</tr>
<tr>
<td>Q4</td>
<td>13731</td>
<td>3969</td>
</tr>
<tr>
<td>Q5</td>
<td>85461</td>
<td>2093</td>
</tr>
<tr>
<td>Q6</td>
<td>136715</td>
<td>6921</td>
</tr>
<tr>
<td>Q7</td>
<td>87661</td>
<td>12673</td>
</tr>
<tr>
<td>Q8</td>
<td>88925</td>
<td>3208</td>
</tr>
<tr>
<td>Q9</td>
<td>117297</td>
<td>7942</td>
</tr>
<tr>
<td>Q10</td>
<td>91085</td>
<td>4174</td>
</tr>
</tbody>
</table>

Table C.1: The results of the experiments of processing algorithms over fragmented XML tree.

*Waiting time.* The ratio of the waiting time of three algorithms, EPP : PP : TP, was 1 : 1.94 : 37.52. Our experiments showed that the partial processing approach absolutely overcame the stream processing approach according to the waiting time criterion. There were two reasons for this explanation: (1) TP does not use the parallelism like PP and EPP; (2) there are many communication between sites. In cases of Q3, Q7 and Q10, although the processing and communication cost of TP was lower than PP’s
but the waiting time was still higher. \textbf{EPP} overcame \textbf{PP} according the waiting time criterion as the ratio was about 50\% and being better in all test cases. Especially in cases \textbf{Q7} and \textbf{Q10}, there were a lot of unnecessary operations for \textbf{PP} but they were restrained by preprocessing over indexes in \textbf{EPP}.

\textit{Processing and Communication Cost}. The ratio of the processing and communication cost of three algorithms, \textbf{EPP} : \textbf{PP} : \textbf{TP}, was 1 : 1.77 : 2.75. The communication cost made the cost of \textbf{TP} was higher than \textbf{PP} in general. However, the unnecessary operations types 1 made the total cost of \textbf{PP} was higher than \textbf{TP} in several cases (\textbf{Q3}, \textbf{Q7} and \textbf{Q10}). By restraining unnecessary operations, the cost of \textbf{EPP} was always the lowest in all test cases among three algorithms.

Our experiments showed that our \textbf{EPP} algorithm was the best among the introduced algorithms according the waiting time and the processing and communication cost criteria.
Publications


References


