UNIVERSITY OF CALIFORNIA
RIVERSIDE

Query Answering in Data Integration Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Mariam S. Salloum

March 2012

Dissertation Committee:

Dr. Vassilis J. Tsotras, Chairperson
Dr. Walid Najjar
Dr. Michalis Faloutsos

The Dissertation of Mariam S. Salloum is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am so happy that I can finally thank some important people that have helped me through this journey. I would like to thank my advisor Dr. Vassilis J. Tsotras for his support, guidance, and enlightening discussions throughout my PhD. He was very supportive and encouraging during the early stages of research and helped me maintain focus on ideas leading to my dissertation. I am grateful for this support these past five years and for the freedom he gave me to pursue various projects. I would also like to thank my other committee members, Dr. Walid Najjar and Dr. Michalis Faloutsos, for providing insightful comments and career advice.

Special thanks to Divesh Srivastava and Luna Dong for hosting me as an intern at AT&T Research Labs. I truly enjoyed my visit to AT&T Research, and I have throughly enjoyed our collaboration since that time. Thank you for introducing me to the area of data integration and for demonstrating how to develop research ideas.

Most importantly, I am thankful to my parents who always found a way to encourage me to accomplish my dreams.

I would like to dedicate this Doctoral dissertation to my parents for their continued support, encouragement, and constant love.

ABSTRACT OF THE DISSERTATION

Query Answering in Data Integration Systems

by

Mariam S. Salloum

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2012
Dr. Vassilis J. Tsotras, Chairperson

This dissertation examines several important problems dealing with building database systems that support query processing and data integration.

The first part of this work focuses on data integration systems and the challenges faced in querying a distributed set of data sources. An Online Answering Systems for Integrated Sources (OASIS) is presented which considers source coverage, overlap, and cost to order source accesses such that answers are returned as soon as possible. The first component of OASIS is a fast and scalable method for estimating source overlaps. The second component considers two dimensions of source ordering, static and dynamic. The last component applies several heuristics to select additional overlap statistics to be computed with the goal of obtaining a better source ordering.

While the first part of the dissertation focuses on multi-source query answering, the second and third part of the dissertation focuses on single-source query answering. The second part of this work proposes a candidate document ordering strategy for a single-source query answering to return answers as fast as possible. Two optimization

problems are considered: optimal ordering and selection of candidate documents. The first problem deals with finding a sequence of documents which minimize the time to first k matches. The second problem deals with finding a subset of documents that maximize the expected number of document matches for a given upper bound on total processing time. The objective functions of the two optimization problems are expressed in terms of two parameters, the probability of a query having a match in a document and the expected document processing time. These two optimization problems are considered for applications which contain inter-document precedence constraints which restrict the order in which candidate documents must be processed.

The third part of the focuses on processing XML queries, and proposed a unified method for solving three important problems in XML structural matching, namely, Filtering, Query Processing, and Tuple-Extraction. The queries and XML documents are represented using a sequential encoding, referred to as Node Encoded Tree Sequences (NETS). The unified solution for the three problems is composed of two procedures, subsequence matching and structural matching, which can be executed concurrently or sequentially depending on the problem. The solution for subsequence matching is based on the dynamic programming recurrence relation for the Longest Common Subsequence (LCS) problem. For structural matching, a new necessary and sufficient condition is presented which provides a simple verification procedure. In addition to using a unified framework, (for easier implementation and maintenance), experimental results show that the proposed algorithms outperform state of the art approaches for the three XML processing problems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the explosion of information provided by World Wide Web (WWW) and scientific databases, efforts have been focused on building efficient, scalable, and integrated database systems. Whether the database is locally stored or distributed over a network (or cloud), whether the database is static or being acquired through a stream, a key optimization factor of such a system is *query answering*. This presents several challenges and opportunities for optimizing query answering when considering a centralized database (or data sources) or a distributed set of data sources. This dissertation spans both dimensions of query answering.

The first part of this dissertation focuses on data integration systems, which consists of a large number of heterogeneous and independent data sources. Examples of such systems include collaborative scientific projects (specifically in biology), digital libraries, and the WWW. Its important to provide users with a unified view of these independent data sources, as well as, provide efficient and scalable query answering solutions. Traditional databases have a standard compilation and execution strategy. First,

the query optimizer statically compiles the query into a plan, then selects the best plan based on the system's knowledge about execution costs, data distribution, etc. Then, a query execution engine is invoked to execution the query plan. This paradigm assumes that a large and fairly accurate set of statistical information is available to the query optimizer. In traditional database settings, this assumption may be satisfied, however, this assumption cannot be made for data integration systems because sources are autonomous and distributed and the system lacks statistical information about their content.

The second part of the dissertation addresses challenges in query answering over a single data source (or a centralized database). A single data source may contain a large set of structured or unstructured documents. It is important to consider multiple objects in query answering, including time to first result, total processing time, and the throughput of the system. The first objective minimizes the time to the first few results, and hence must consider the properties and expected cost of the candidate document and the order in which to evaluate candidate documents. The second objective depends on the query execution (or query processing algorithm) utilized to query a candidate document. Lastly, the throughput of the system may depend on the properties of the documents, queries and the query processing algorithm employed, and its important to devise an approach that does not perform poorly when properties of the database vary.

1.1 Contributions of this Dissertation

Chapter 2 of the dissertation presents and evaluates an Online Query Answering Systems for Integrated Sources (OASIS). A data integration system offers a uniform

interface for querying a large number of autonomous and heterogeneous data sources. Individual sources typically contain only a fraction of the possible answer tuples to a query, thus the data integration system must probe several sources to obtain a complete or sufficient set of answers. A basic component in such a system is ordering sources accesses to maximize the likelyhood of obtaining query answers quickly. Source ordering becomes more challenging when there is overlap (or redundancy) between data sources. Since data sources may overlap in the content they provide, the on-line query answering systems must integrate the results and remove redundancy before displaying results to the user. Moreover, the system must avoid retrieving the same set of answers from multiple sources in order to increase query performance (time or access cost) and system availability. Sources are autonomous and will not publish such overlap statistics, thus the system will be responsible for generating the overlap statistics between sources. Motivated by the dynamic and continually evolving nature of data integration systems, we introduce an on-line and incremental approach for generating source overlap statistics using a partial set of approximated overlap statistics and the Maximum Entropy (MaxEnt) estimation model. Furthermore, we present two source access plans, namely, *static ordering* and *dynamic ordering*. Static ordering accesses sources in a greedy fashion based on a source's residual coverage. Dynamic ordering, on the other hand, incorporates new overlap statistics in to the MaxEnt model to generate better access plans on-the-fly for the un-accessed sources. A statistics selection strategy is presented, which suggests the computation and inclusion of additional overlap statistics in to the MaxEnt problem formulation. Two ordering strategies that utilize the statistic selection procedure are presented, namely, Static+ and Dynamic+ ordering. The presented ordering strategies can

be viewed as having two dimensions. The first dimension controls the adaptability of the orderings, thus from this we derived two adaptable algorithms Dynamic and Dynamic+. The second dimension controls whether *additional* statistics are incorporated. From this we derive two orderings, Static+ and Dynamic+. The three components, overlap estimation, statistic selection, and source ordering compose the online query answering system OASIS.

Chapters 3 and 4 of this dissertation examines the query answering problem for a single-source environment. Chapter 3 presents a candidate document ordering strategy to return answers as fast as possible. Two optimization problems, optimal ordering and selection of candidate documents for query answering. The first problem deals with finding a sequence of documents which minimize the time to first k matches, for some constant k which is less than the total number of matches. The second problem deals with finding a subset of documents that maximize the expected number of document matches for a given upper bound on total processing time. The objective functions of the two optimization problems can be expressed in terms of two parameters, the probability of a query having a match in a document and the document processing time. For some applications, inter-document precedence constraints exist which restricts the order in which candidate documents must be processed; such constraints can be modeled as chains. Thus, the two optimization problems are considered for applications with and without precedence document constraints. A a polynomial-time algorithm is presented for the document ordering problem. The optimal solution to the selection problem may take exponential time, consequently, a heuristic algorithm is devised, for which experimental evaluations show that it is a good approximation of the optimal solution.

Chapter 4 presents a unified method for solving three important problems in XML structural matching: Filtering, Query Processing, and Tuple-Extraction. The the queries and XML documents using a sequential encoding, referred to as Node Encoded Tree Sequences (NETS). The unified solution for the three problems is composed of two procedures, subsequence matching and structural matching, which can be executed concurrently or sequentially depending on the problem. The solution for subsequence matching is based on the dynamic programming recurrence relation for the Longest Common Subsequence (LCS) problem. For structural matching, a new necessary and sufficient condition is presented which provides a simple verification procedure. An efficient algorithm is presented for the XML filtering and tuple-extraction problems, where subsequence and structural matching are performed concurrently, referred to as Forward-Match. The Forward-Match algorithm utilizes a novel recurrence relation for subsequence and structural matching. This recurrence relation combines the new necessary and sufficient condition and the LCS recurrence relation. For the query processing problem we present an efficient algorithm (referred to as Backward-Match), that performs subsequence and structural matching sequentially and utilizes a compact graph representation of all potential subsequence matches. In addition to using a unified framework, (for easier implementation and maintenance), experimental results show that the proposed algorithms outperform state of the art approaches for the three XML processing problems.

1.2 Outline

The following three chapters (Chapter 2 to 4) will elaborate on the problems outlined in this chapter. Finally, Chapter 5 concludes the dissertation and discusses directions for future research. Parts of this dissertation have been published in conferences and workshops. In particular, the candidate document ordering problem outlined in Chapter 3 is described in [72], and the XML filtering algorithm and the FPGA-based XML filtering algorithm described in Chapter 4 are described in [71] and [53, 54], respectively.

Chapter 2

On-line Query Answering

2.1 Introduction

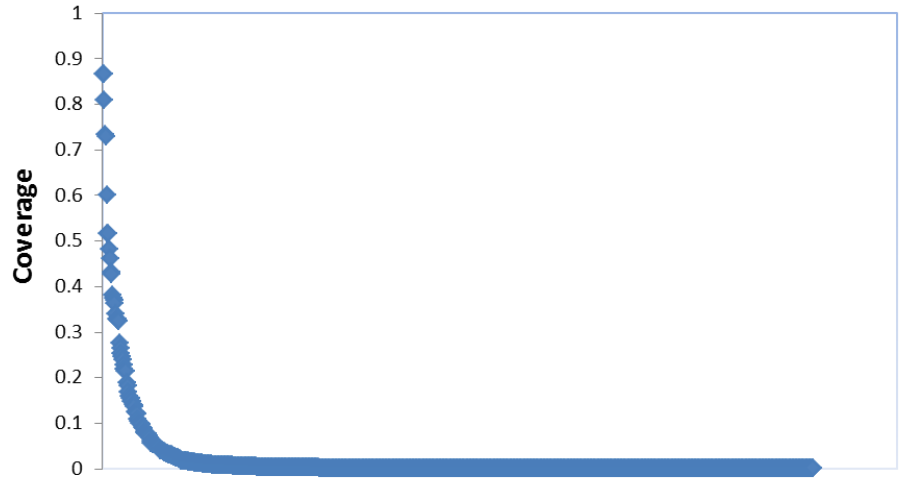
A vast number of autonomous data sources are available on the Internet. Data Integration Systems (or mediator systems) offer users a uniform interface for querying a large and distributed set of data sources [27, 28, 29, 56, 9, 74, 38]. For example, consider a data integration system providing information about books from data sources on the World-Wide-Web (WWW). There are numerous sources on the WWW providing listing of books, their authors, editors, etc., and other sites providing other information such as reviews or summaries for selected books. Suppose a user poses a query to find all books written by Jane Austin and their respective reviews. No individual source can answer this query completely. However, by joining data from multiple source queries can be answered completely.

A Data Integration Systems generally consists of three main components: query formulator, query optimizer and execution engine. Given a user query, the query for-

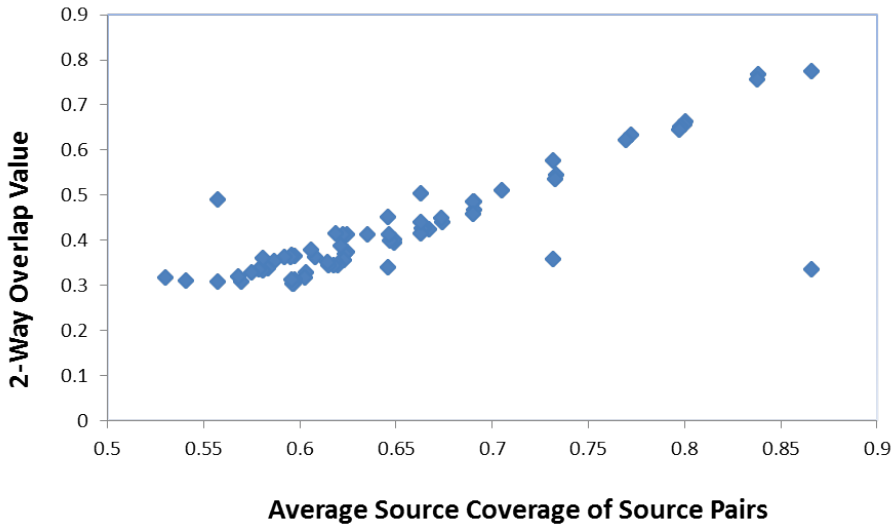
mulator rewrites the query using a mediated schema and generates a set of query plans. The query plans specifies the order of source accesses and how the data is combined to obtain the answer to the query. At the optimizer level, most systems have focused on minimizing the cost to obtain all query answers. In data integration systems, where the amount of information, as well as, the amount of query answers are very large, a user is more interested in quickly receiving a large fraction of answers rather than waiting for a long time to receive the full set of answers. Ordering data source accesses can optimize the rate at which answer tuples are received. This is a challenging task because data sources have varying coverage and may overlap in the content they provide. Furthermore, there are various cost metrics (access agreements, latency, etc.) associated with accessing and querying each data source which must be considered in query answering. Hence, an important optimization problem is ordering source accesses in decreasing order of their utility, which is defined as the number of new query answers obtained from the source. Query execution can be aborted as soon as the user has received a satisfactory set of query answers, or until full set of answers are returned.

A collection of computer science books was extracted from AbeBooks.com, a listing-service website that integrates information from online bookstores. The data collection includes 1028 bookstores (sources) and approximately 1256 records/ objects. Figure 2.1(a) shows the coverage (defined as the faction of objects provided by a given source) of the individual data sources plotted in descending order of coverage cardinality. As shown, the coverage of data sources vary, where some sources have very high coverage, while the majority have relatively small coverage. The 2-way overlaps between data source pairs is plotted in Figure 2.1(b). Its obvious selecting and ordering sources in descending order

of their coverage may not be the best strategy since the residual coverage (the number of new tuples contributed to the result set) may be small. The source ordering policy



(a) Source Coverage



(b) 2-Way Source Overlap (for largest 20 source pairs)

Figure 2.1: Plot of source coverage and overlap for the AbeBooks.com data collection.

must consider several factors in ordering data sources. First, the query answering system should have probabilistic knowledge about degree of relevance (or coverage) of each

data source to the query. Second, the system should have probabilistic knowledge about the size of intersection of answer sets among data sources for that query (i.e. overlap between data sources). Third, access cost of querying a particular data source. Sources may publish/disclose coverage and cost information. However, since data sources are autonomous and decentralized, source overlaps are not directory available to the query answering system.

The amount of information necessary to specify source overlaps is exponential in the number of sources. The key challenge in learning or estimating such statistics is keeping the needed statistics under control, since naive approaches can become infeasible very quickly. Furthermore, the information provided by the data sources usually evolves over time, thus, the data integration system must maintain up-to-date and accurate statistics concerning the overlaps between data sources. The architecture of our system is depicted in Figure 2.2.

An **O**nline **A**nswering **S**ystem for **I**ntegrated **S**ources (**OASIS**). Statistics which describe the cost, coverage, and partial overlaps between data sources are generate as input to the system. First, the **Source Selection** component chooses a subset of sources which are relevant to the query. Second, the **Overlap Estimation** component estimates the overlap between all subsets utilizing the previously generated coverage and partial overlap statistics. Third, the **Source Ordering** component orders the set of sources such that query answers are reported as fast as possible. Using a static policy, sources will be accessed in the order specified. Using a dynamic policy, the next source in the ordering is accessed and its coverage and union statistics are computed on-the-fly. These

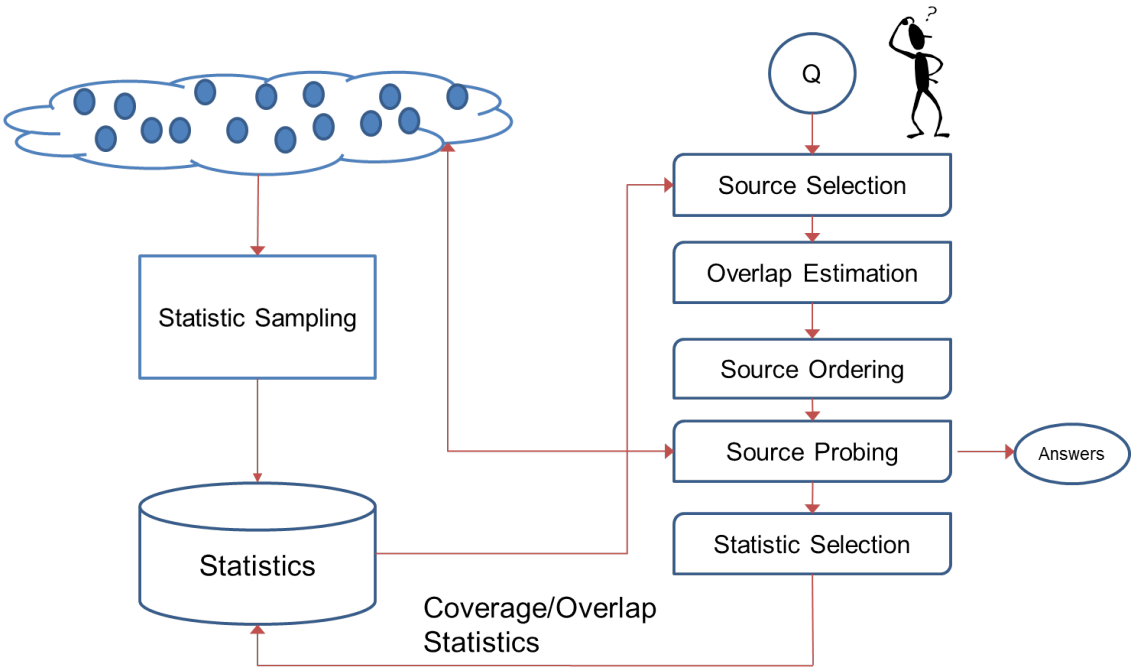


Figure 2.2: Data Integration System Architecture

new overlap statistics are added to the problem definition and are used to re-estimate the overlaps between data sources. The intuition here is that the additional statistics will improved the estimated overlaps and will contribute to a better source ordering of the un-probed sources. Lastly, the **Statistic Selection** component will evaluate and select a small set of overlap statistics to be incorporated into the problem formulation. These overlap statistics can be requested from a Third Party statistic server, or computed by the data integration system. In Section 2.2 the various challenges are outlined and the sub-problems defined. Section 2.3 discusses the overlap estimation component. In Section 2.4 the source ordering problem is defined and a greedy algorithm is presented. Two variations of the source ordering algorithm is presented, static ordering and dynamic ordering. In Section 2.5 a statistic selection strategy is presented. Extensive experimental results are presented in Section 2.7. Section 2.6 presents related work on data integration

systems which consider overlap between data sources. In Section 2.8 final remarks and future work are discussed.

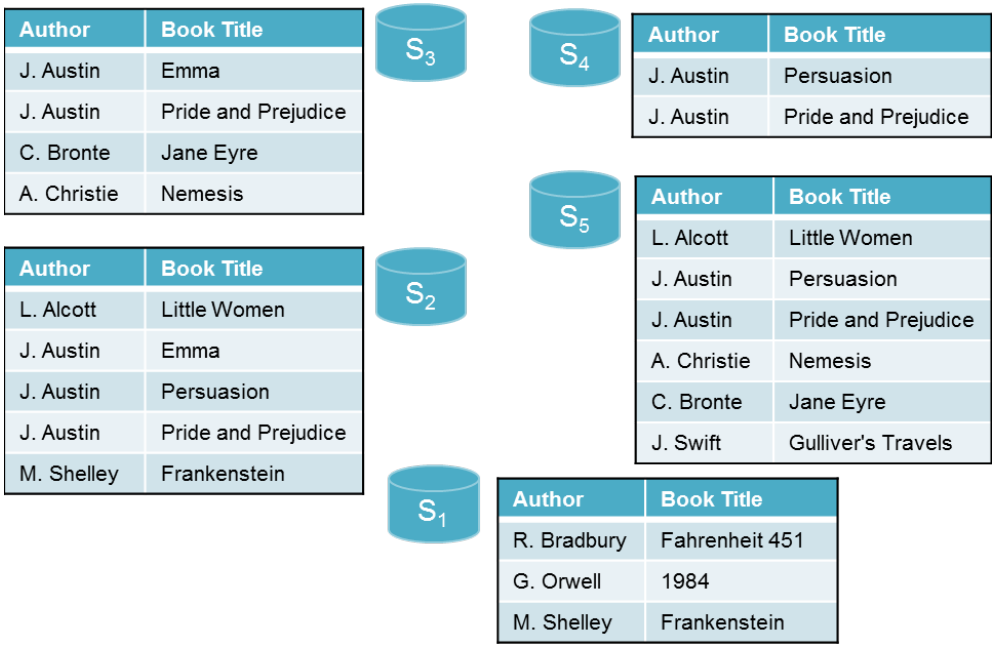
2.2 Problem Statement

This section presents definitions that will be used throughout the chapter and present the problem statements. A user is most interested in quickly receiving a large fraction of answers rather than waiting for a long time to receive the full set of answers. Hence, in order to report answer tuples as fast possible, we must consider source coverage, cost, and the overlap between data sources. Definition 2.1 defines source coverage in terms of probabilistic information. Consider Figure 4.1 and a general SELECT * query. Data source source S_1 has a coverage of $P(S_1) = 30\%$. Definition 2.2 defines overlap of a set of sources as the size of their intersection. For example, the overlap of S_1 and S_2, denoted as $P(S_1 \cap S_2)$, is 10%.

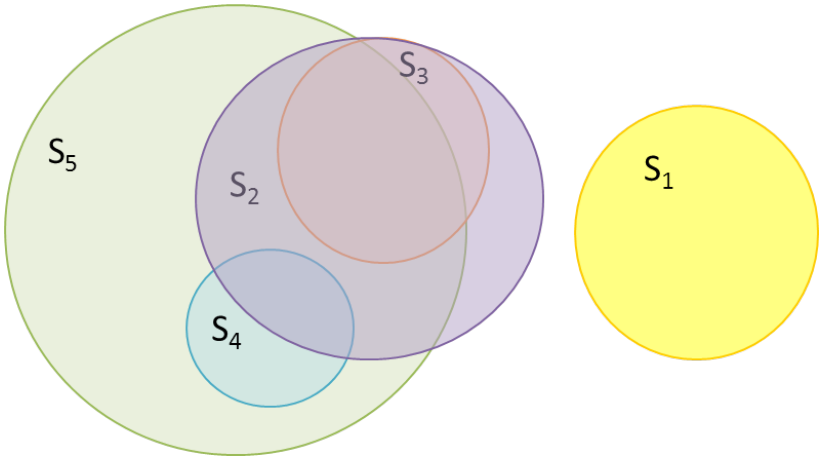
Definition 2.1. *(Coverage) The coverage of source $S_i \subseteq S$, denoted as $P(S_i)$, w.r.t. a query Q is the expected fraction of answer tuples obtained from source S_i.*

Definition 2.2. *(Overlap) The overlap of a set of sources $S = \{S_1, S_2, \ldots S_n\}$, denoted as $P(S_1 \cap S_2 \cap \ldots S_n)$ or $O(S)$, is the fraction of of answer tuples obtained from the intersection of the data sources in the set.*

Example 2.1. ***Motivational Example*** *Consider the 5 sources shown in Figure 4.1. A naive ordering approach would select the sources in descending order of their coverages. The ordering generated based on this policy is S_5 S_2 S_3 S_1 S_4. However, this is not the best*



(a) Data Sources Example



(b) Data Sources Venn Diagram

Figure 2.3: Motivational example for overlap consideration in query answering systems

ordering when we consider the overlap between these sources. The overlap between data sources is represented by the venn diagram in Figure 2.3(b). Source S_5 should be selected first since it has the highest coverage. Even though S_2 has the second largest coverage, it overlaps with S_5 and hence its residual coverage (i.e. the number of new objects S_2

contributes which are not provided by S_5) is only two tuples. An ordering that considers residual based on the source overlaps is S_5 S_1 S_3. Observe that only three sources were needed to obtain full coverage. Based on this example, its obvious that we must consider more than coverage in selecting and ordering data sources for query answering.

The focus of this chapter is to present a efficient and scalable solution for answering a query over multiple data sources. Throughout the rest of this chapter, the solution to query answering will be presented for a query of the form: find all answer tuples from all sources. Future work will examine extensions to support more complex queries. The next section presents the Overlap Estimation Component.

2.3 Overlap Estimation

In this section, we present the **Overlap Estimation** component which is responsible for generating source overlap estimates. The estimated source overlaps will be utilized by the Source Ordering component to generate an ordering of data sources. Generally, data sources will publish or disclose their coverage information, so we will assume such probabilistic knowledge is available or can be obtained easily. The coverage of a data source S_i is denoted as $\mathrm{P}(S_i)$. Furthermore, we shall assume that a small number of source overlaps are provided (whose value are either precise or estimates obtained via sampling). The set of overlap statistics can describe 2-Way upto n-Way overlaps. The Overlap Computation problem is defined more formally as follows:

Problem 2.1. ***(Overlap Computation Problem)***

Let $S = \{S_1, S_2, \ldots, S_n\}$ be a set of n data sources, and L be a list of all possible source

subsets of S. Given set S, coverage P(S_i) $\forall$ $S_i \in S$, and the overlap O(L_i) $\forall$ $L_i \in \bar{L}$, where $\bar{L} \subset L$. Compute the following: $\forall$ $\bar{S} \subseteq S$, where $\bar{S} \notin \bar{L}$ find O($\bar{S}$).

We use entropy maximization and incorporate all available statistical information (represented as a set of linear inequality constrains) to define the mathematical problem. The MaxEnt framework is useful for several reasons. First, the MaxEnt essentially provides the highest likelihood on the probability distribution taking into account the provided statistics, and makes no additional assumptions beyond those provided statistics. Second, the MaxEnt framework changes smoothly with the addition of new statistics, or the modification of previously provided statistical information.

2.3.1 Problem Formulation

The objective function is to maximize entropy under the known coverage and overlap statistics. The solution will provide the most likely distribution under the given constraints. Consider the following example.

Example 2.2. *Consider a set of five sources S = A,B,C,D,E. The coverage of all five sources and three overlap statistics are given below:*

$P(A) = .50$ $\quad P(A \cap B) = .30$

$P(B) = .40$ $\quad P(C \cap D) = .18$

$P(C) = .35$ $\quad P(A \cap C \cap D) = .08$

$P(D) = .20$

$P(E) = .05$

The set of equality constraints can be generated using 2^n variables. These variables, called K-pos variables, express the inclusion of k sources and the exclusion of (n-k)

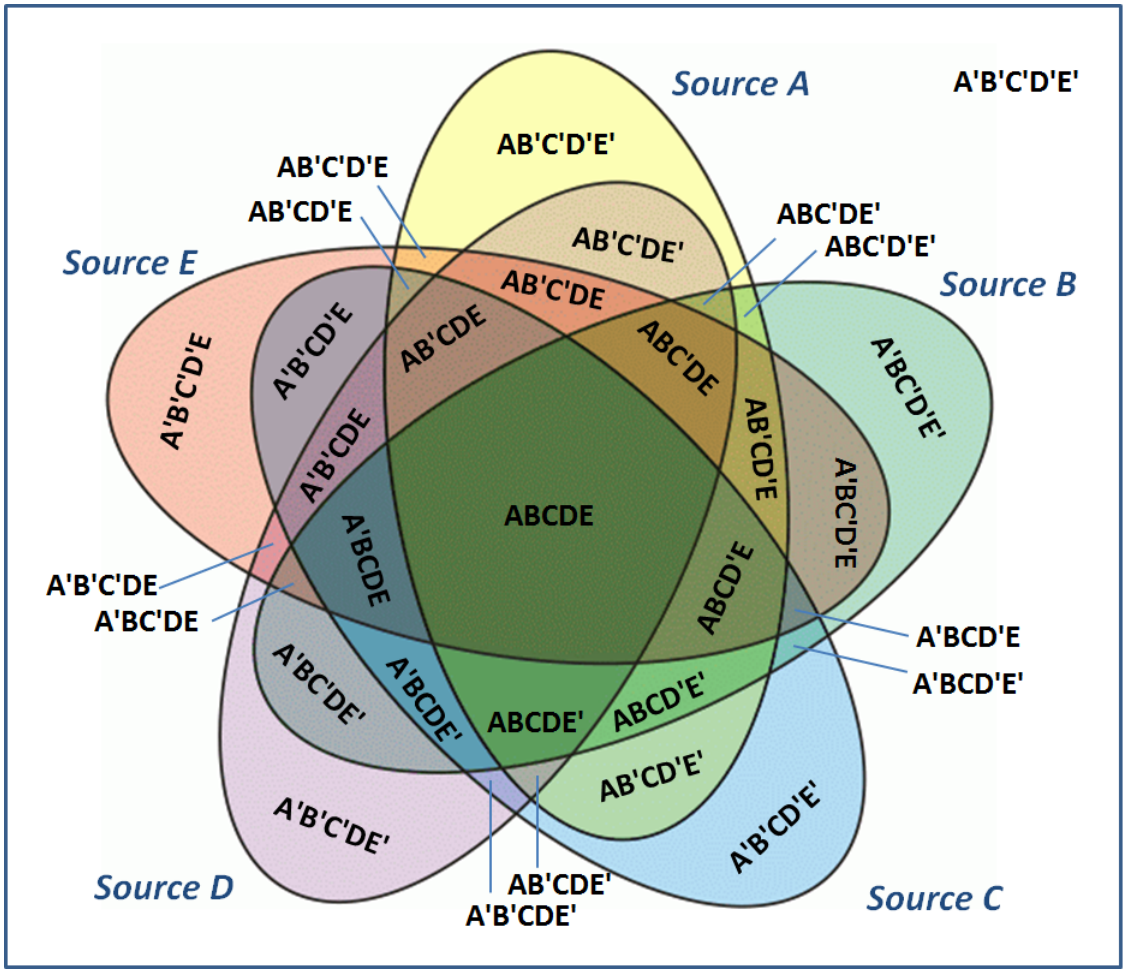


Figure 2.4: Five source venn diagram

sources, as defined below.

Definition 2.3. *(K-pos Variable:) A k-positive (k-pos) variable denotes a variable which contains k positively expressed sources. This variable denotes the area in the venn diagram that is exclusive to k positively expressed sources.*

Consider the set of five sources given in Example 2.2. A Venn diagram, shown in Figure 2.4, illustrates all possible intersections between the five sources. Figure 2.2 shows 2^n variables; there are five sources hence 2^5 (or 32) variables are defined. Each area in the

Venn diagram corresponds to a k-pos variable. For example, variable **A B** C' D' E' is a 2-pos variable and represents the inclusion of sources A and B and the exclusion of sources C, D, and E. Essentially, this variable refers to the area (and the tuples contained within this area) in the venn diagram that is exclusive to sources A and B. Similarly, variable **A B C** D' E' is a 3-pos variable because it represents the inclusion of three sources A, B, and C. Observe that variable A' B' C' D' E' (a 0-pos variable) is defined to note the area that is not covered by any source.

The provided coverage and overlap statistics (which may be approximated or stale) are formulated as linear inequality constraints. For example, given that the coverage of source A is 5%, then the constratint can be specified as the sum of $2^{nSources-1}$ variables where source A is positively expressed.

$$\text{P(A)} = .50 = \boldsymbol{A}B'C'D'E' + \boldsymbol{AB}C'D'E' + \boldsymbol{A}B'\boldsymbol{C}D'E' + \boldsymbol{A}B'C'\boldsymbol{D}E' + \boldsymbol{A}B'C'D'\boldsymbol{E} + \boldsymbol{ABC}D'E' + \boldsymbol{A}B'\boldsymbol{CD}E' + \boldsymbol{AB}C'D'\boldsymbol{E} + \boldsymbol{A}B'\boldsymbol{C}D'\boldsymbol{E} + \boldsymbol{A}B'C'\boldsymbol{DE} + \boldsymbol{ABCD}E' + \boldsymbol{ABC}D'\boldsymbol{E} + \boldsymbol{A}B'\boldsymbol{CDE} + \boldsymbol{ABCDE}$$

Given an overlap statistic which specifies that the overlap of sources A and B are 3%, then the costraint specifiying this overlap can be expressed as the sum of $2^{nSources-2}$ variables where both source A and B are positively expressed.

$$\text{P(A} \cap \text{B)} = .03 = \boldsymbol{AB}C'D'E' + \boldsymbol{ABC}D'E' + \boldsymbol{AB}C'\boldsymbol{D}E' + \boldsymbol{AB}C'D'\boldsymbol{E} + \boldsymbol{ABCD}E' + \boldsymbol{ABC}D'\boldsymbol{E} + \boldsymbol{AB}C'\boldsymbol{DE} + \boldsymbol{ABCDE}$$

This mathematical problem is defined in terms of all possible source intersection combinations, hence, it requires the definition of 2^n variables, where n is the number of

data sources. This number quickly becomes unmanageable in terms of CPU time required to solve the problem and memory required to define and solve the problem. Observe that the number of variables needed to define the problem should not exceed the number of objects provided by the union of all data sources. A area on the Venn diagram that is empty (provides no answer tuples) does not require a the definition of a k-pos variable. Moreover, the number of objects (or answer tuples) is most likely less than the 2^n variables needed to represent the problem. Thus, our approach will make use of this observation and try to reduce the number of variables.

The rest of this section will present an incremental method of generating the estimated cardinality (or size) of the areas on the Venn diagram. Instead of defining 2^n variables, approach will incrementally compute the overlap between data sources. The approach will refine the resolution of estimated areas by determining large areas and estimating those areas with higher resolution. Resolution refers to the number of variables that are used to define the coverage/overlap of the area of interest. The approach can be summarized as follows: Initially, we assume that intersections between 2 or more sources are empty (i.e. only 1-pos variables are defined), unless an k-way overlap statistic is provided which is expressed using a k-pos variable. The problem is formulated as a set of inequality constraints and solved. Once variable estimates are generated, each k-pos variable is expanded to a (k+1)-pos variable if its deemed large (among top k-pos variables, above a threshold, etc.). The expansion step is repeated until no additional variables are introduced.

Example 2.3. *Reconsider the set of five sources in Example 2.2. The set of constraints*

can be expressed using 1-pos variables corresponding to source coverage statistics and k-pos variables corresponding to k-way overlap statistics, as follows:

$P(A) = \boldsymbol{A}B'C'D'E' + \boldsymbol{AB}C'D'E' + \boldsymbol{A}B'\boldsymbol{CD}E'$

$P(B) = A'\boldsymbol{B}C'D'E' + \boldsymbol{AB}C'D'E'$

$P(C) = A'B'\boldsymbol{C}D'E' + A'B'\boldsymbol{CD}E' + \boldsymbol{A}B'\boldsymbol{CD}E'$

$P(D) = A'B'C'\boldsymbol{D}E' + A'B'\boldsymbol{CD}E' + \boldsymbol{A}B'\boldsymbol{CD}E'$

$P(E) = A'B'C'D'\boldsymbol{E}$

$P(A \cap B) = \boldsymbol{AB}C'D'E'$

$P(C \cap D) = A'B'\boldsymbol{CD}E' + \boldsymbol{A}B'\boldsymbol{CD}E'$

$P(A \cap C \cap D) = \boldsymbol{A}B'\boldsymbol{CD}E'$

Essentially, we are assuming that a k-pos variables, where k $>$ 1, have zero cardinality. When solving the MaxEnt problem, the tuples provided by this area will be distributed among its (k-1)-pos variables. If the k-pos variable is $AB'CDE'$, then its 2-pos variables are $ABC'D'E'$, $AB'C'D'E$, and $A'BC'D'E$ will absorb its coverage. If the sum of these (k-1)-pos variables is large, then it may indicate that the k-pos variable may be large or contain some significant portion of tuples. The constraints will be expanded to contain k-pos variables deemed large.

The constraints for the MaxEnt problem were previously defined as equality constraints, but since only a few variables are used initially to define the set of constraints the MaxEnt problem cannot generate a feasible solution. A delta is added/subtracted from constraint values to create an upper/lower bound. This delta will be reduced as

more variables are added to the constraints.

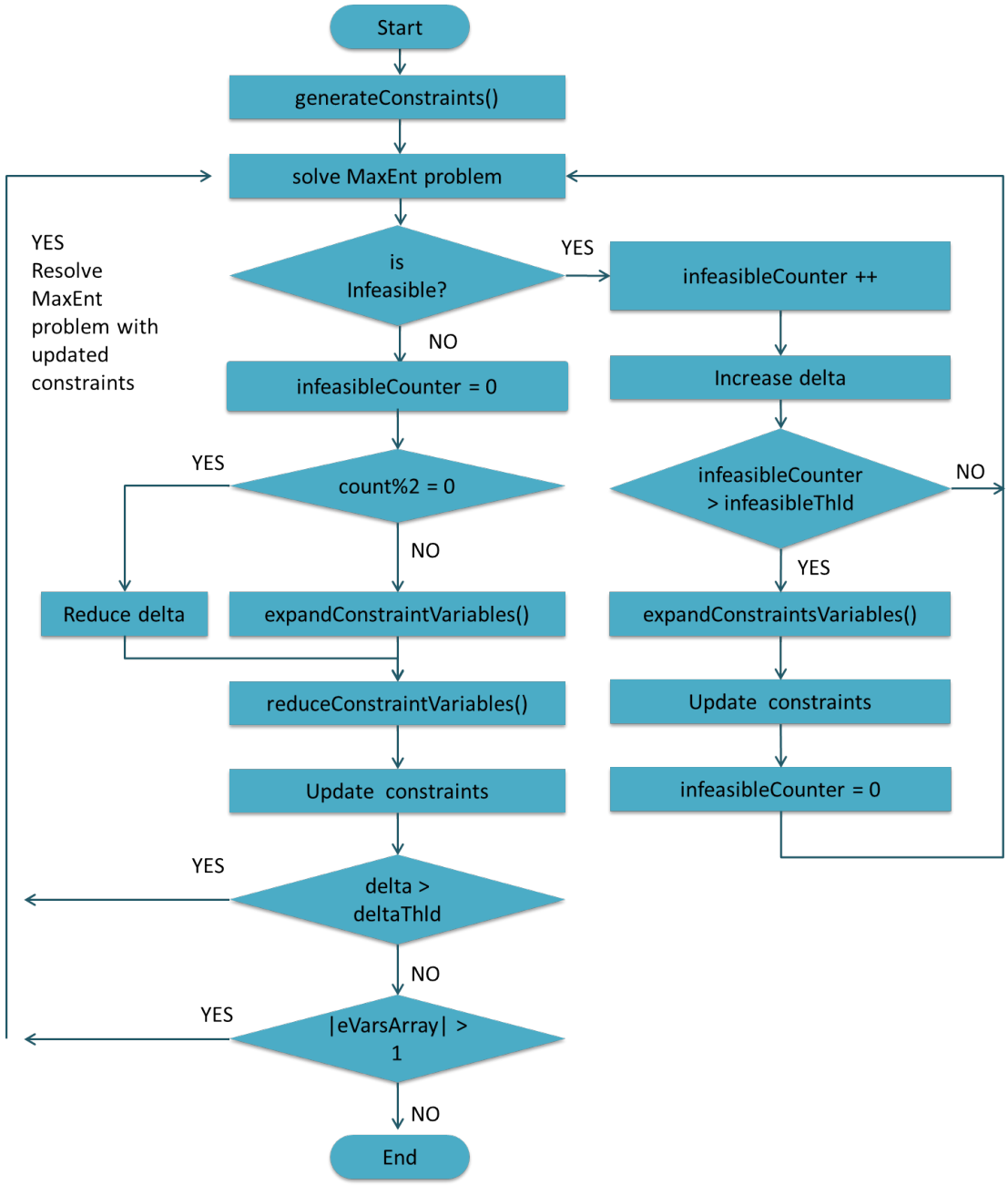


Figure 2.5: Algorithm 2.1 Data Flow Diagram

A data flow diagram of the Overlap Estimation component is shown in Figure

Table 2.1: Algorithm 2.1 Parameters

delta	Delta +/- to constraints to create upper/lower bound
deltaOverall	Delta +/- to constraint describing total coverage
deltaThld	Threshold on delta used to terminate loop, used in Algorithm 2.1
constraintThld	Lower bound on constraint value used for deciding whether variables should be expanded, used in exapandVariables(.) procedure
variableThld	Lower-bound on variable value, used in procedure reduceVariables(.)
infeasibleThld	Max number of allowed iterations which result in infeasible solution before expandVariables(.) procedure is invoked.
nMaxExpanded	Upper-bound on number of expanded variables at each iteration.

2.5. The method receives source coverage and a partial list of overlap statistics as a input. These statistics are represented as inequality constraints using k-pos variables, where k corresponds to the number of sources expressed by a given statistic. A non-linear solver is invoked to solve the MaxEnt problem. Two methods are used to resolve infeasible solutions. First, a delta added/subtracted from the upper/lower constraint bounds is increased. Second, a set of variables are created and added to the constraints if its k-pos variables are deemed large. The details of the Overlap Estimation Component is given by Algorithm 2.1. Table 2.1 lists all parameters of Algorithm 2.1.

The Overlap Estimation Component, given by Algorithm 2.1, is composed of several procedures. Given source coverage and overlap statistics, Procedure generateConstraints(.) is invoked to generate the minimal set of variables. The MaxEnt problem is solved for the initial set of constraints. If the problem is determined to be feasible (a distribution under the given constrains was found), then the next step is to expand the problem, either by reducing the delta upper/lower bounds on the inequality constraints or

Algorithm 2.1: overlapEstimation()

Input: **statisticArray: sourceList (list of data sources), overlapValue (double)**

Output:
varArray */* Array of variables containing 2-pairs variableId, value*/*

```
1  constraintArray ← Procedure generateConstraints;
2  while (delta > deltaThld OR eVarsArray ≠ NULL ) do
3  |  varArray ← solve MaxEnt problem;
4  |  if infeasible solution then
5  |  |  infeasibleCounter ++;
6  |  |  delta = delta * 2;
7  |  |  if infeasibleCounter > infeasibleThld then
8  |  |  |  eVarsArray ← Procedure expandConstraints update
   |  |  |  constraintArray to include variables in eVarsArray. ;
9  |  |  |  infeasibleCounter = 0;
10 |  |  end
11 |  else
12 |  |  rVarsArray ← Procedure reduceConstraints()        /* Alternate
   |  |  between expanding variables and reducing delta  */
13 |  |  if (count%2 = 0) then
14 |  |  |  eVarsArray ← Procedure expandConstraints ;
15 |  |  else
16 |  |  |  delta = delta/2 deltaOverall = deltaOverall/2
17 |  |  end
18 |  |  update constraintArray exclude/include variables in
   |  |  rVarsArray/eVarsArray
19 |  end
20 end
```

by adding more variables to the problem definition. The Procedure expandConstraints(.) is invoked to determine the set of variables to add to the problem definition. Procedure expandConstraints(.) selects a small number of k-pos variables (that are not currently defined) to create and include in the problem formulation using information about the variables (k-1)-pos variables.

The procedure examines all variables in varArray. Each k-pos variable is expanded to generate a set of (k+1)-pos variables. Each (k+1)-pos variable is considered

Procedure generateConstraints

Input:
constraintArray */*Each entry is a 3-tuple: sourceBitset(bitset), variables(int array), value(double)*/*
nConstraint */*number of constraints*/*

Output:
constraintArray

```
1  foreach (i=0 to nConstraints) do
2  |  S ← constraintArray[i] ;
3  |  S.variables ← create variable;
4  |  foreach (j=0 to nConstraints & i ≠ j) do
5  |  |  S̄ ← constraintArray[j];
6  |  |  if ( S̄.sourceBitSet.isSubset(S.sourceBitSet) ) then
7  |  |  |  S̄.variables ← add S.variables;
8  |  |  end
9  |  end
10 end
```

as a candidate. Two conditions must be satisfied inorder for the (k+1)-pos variable to be added to the constraints. The first condition requires that the sum of the k-pos variables corresponding to (k+1)-pos variable is large enough. The second condition considers all constraints, if a constraint is a subset of k+1-pos variable (but not a subset of k-pos variable) and has a value less than a threshold (or less than the k-pos variable which generated (k+1)-pos variable), then the (k+1)-pos variable is discarded. Otherwise, if the two conditions are satisfied, the (k+1)-pos variable is added to the set of the set of expanded variablesVarsArray.

Procedure reduceConstraints selects a small number of variables to remove from the problem formulation. These variables are selected based on their 'estimated' value from previous MaxEnt solution. The intuition here is that variables that are very small will not affect the value of the overlap and hence the ordering of data sources. Since

such variables are not contributing to the accuracy of the result, but do hold space, they should be removed. Procedure generateConstraints(.) generates the set of constraints and their corresponding variables in the initial problem formulation. The accuracy of the overlap estimates generated depends on several factors. First, the number of initial overlap statistics provided as input to the Algorithm 2.1 plays a large role in determining the accuracy of the estimated overlaps. A very small number of initial statistics may not provide sufficient information to accurately estimate the unknown overlaps. Second, the distribution of initial statistics should be varied and not be limited to providing statistics concerning a few sources while omitting others. Third, the number of variables used to formulate the problem (i.e. the variables used to define the constraints describing the coverage and overlap statistics) highly affect the accuracy of the estimates. In our approach, the number of variables is determined by two algorithm parameters. The first parameter (nMaxExpanded), restricts the number of (k+1)-inclusion variables generated and included in the problem formulation on next iteration of the procedure. The second parameter, constraintThld, is a lower bound on constraint values, which prunes candidate expansion variables which correspond to constraints whose value fall below the threshold. These factors will be analyzed in the experimental section to evaluate the accuracy of the unknown source overlaps and consider how these parameters affect the generated source ordering.

Procedure expandConstraints

Input:
varArray: variableId(integer), value(double) */*Variables considered for expansion*/*
expansionVarsArray */*List of candidate expansion variables*/*

Output:
expansionVarsArray */*List of candidate expansion variables*/*
eVarsArray */*Variables to be added to constraints*/*

```
1  flag = true;
2  eVarsArray ← ∅;
           /* Loop through varArray to consider each variable for
   expansion */;
3  foreach ( i = 0 to |varArray| ) do
4      k_posVar ← varArray[i];
5      k+1_posVarArray[]← generate list of expanded variables from
       k_posVar;
6      foreach ( j = 0 to |k + 1_posVarArray| ) do
7          if k+1_posVarArray[j] ! ∃ in expansionVarsArray then
8              flag = true;
9              k_posVarArray ← list of k-pos variables corresponding to
               k+1_posVarArray[j];
                     /* only maintain top 'nMaxExpanded' variables */
10             sum ← compute sum of variables in k_posVarArray;
11             if ( sum qualifies as top nMaxExpanded ) then
12                 l = 0;
13                 while flag ≠ false & l < nConstraints ;
14                 do
15                     if constArray[l].sourceBS k+1_posVarArray[j]
16                     & constArray[l].value ≥ constraintThld then
17                         flag ← false;
18                     end
19                     l++;
20                 end
21                 if flag ! = false then
22                     eVarsArray ← add k+1_posVarArray;
23                     expansionVarsArray ← add k+1_posVarArray;
24                 end
25             end
26         end
27     end
28     remove k_posVar from expansionVarsArray after considering all its
       k+1_pos variables for expansion;
29 end
```

Procedure reduceConstraints

Input:
globalVarList
varArray */∗ Variable array 2-tuple, varID(bitset) and varValue(double) ∗/*

Output:
rVarsArray/∗list of variables to remove from constraints∗/

```
1 variableThld = 1/nObjects;
      /* loop through all variables to remove those that are too
  small */
2 foreach i=0 to nVariables do
3 |   if varArray.get(i).varValue < variableThld then
4 |   |   add varArray.get(i).varID to rVarsArray;
5 |   end
6 end
```

2.4 Source Ordering

Table 2.2: Table of Notations

S	Set of n data sources
$\overline{S}$	Set of probed sources
$\hat{S}$	Set of un-probed sources
S'	Next data source considered for probing
varList$_{\widehat{S}}$	Variables corresponding to un-probed sources
RC($\overline{S}$, S')	Residual coverage of S' w.r.t. $\overline{S}$

The **Source Ordering** component finds an optimal order of sources in which to execute the query so that query answers are returned as fast as possible. The rate at which answers are retrieved can be captured as the area under the curve. Given a permutation $\prod$ of data sources, where $\prod$(i) denotes the i^{th} source in the permutation the area under the curve for permutation $\prod$ is:

$$A(\prod) = \sum_{i=1}^{n} c(S_{\prod(i)}) \times |Q(\cup_{j=1}^{i-1} S_{\prod(j)})| \tag{2.1}$$

Given a set of data sources S $= \{S_1, S_2, \ldots, S_n\}$, an optimal permutation of data sources is one such that for any other permutation $\prod$, we have $A(\prod_{opt}) \geq A(\prod)$.

To obtain the optimal ordering of data sources, we must generate all possible data source permutations, compute the area under the curve, and then select the permutation which maximizes the area. This is obviously not feasible for an on-line system, thus, we propose a simple greedy algorithm given by Algorithm greedySelect. Notations utilized by the algorithm are given by Table 2.2. In the first iteration, the algorithm selects

Procedure greedySelect

Input:
$\prod$
varArray */* Array of variables (2-tuple: varID(bitset) and varValue(double)).*/*
varList$_{\widehat{S}}$ */*List of variables of unprobed sources* $\widehat{S}$*.*/*
Output:
S' */*Next source in the permutation*/*

```
1 foreach each S' ∉ ∏ do
2 |  rc ← getResidualCoverage(∏, S', varArray, varList_Ŝ) ;
3 end
4 let S_max be the source in S which has the highest rc (residual coverage);
5 add S_max to ∏ and remove from S;
6 remove variables corresponding to S_max from varList_Ŝ ;
```

the source with the highest coverage. In subsequent iterations, the residual coverage of each candidate source is computed and the source with the highest residual coverage is chosen. Let $\overline{S} = \{S_1, S_2, \ldots, S_n\}$ denote the set of probed sources at step i. At step i+1, to select the next source to probe, we shall compute the residual coverage provided by each of the sources in $\widehat{S}$ (the set of unprobed sources), i.e. the expected number of new tuples which are not provided by the data sources in $\overline{S}$. We select an unprobed source in $\widehat{S}$ which maximizes the residual coverage. Note, we generate only the next source in the permutation instead of computing the entire ordering of sources. Now, we discuss how to compute the residual coverage of a candidate source (S') with respect to a set of probed sources($\overline{S}$). Residual coverage will be used to determine the best ordering of data sources, thus its important to optimize this operation.

Lets consider a 3-source example. First, we select the first and second best sources, S_1 and S_2. The residual coverage of the third source S_3 can be computed as follows:

$$RC(\neg S_1 \cap \neg S_2 \cap S_3) = |S_3| - |S_1 \cap S_3| - |S_2 \cap S_3| +$$

$|S_1 \cap S_2 \cap S_3|$

In general, the residual coverage of a source S given that n sources $\overline{S} = \{S_1, S_2, S_3, \ldots, S_n\}$ have been selected is

$$RC(\neg \overline{S} \cap S') = |S'| + \sum_{i=1}^{n} \{(-1)^i \sum_{\overline{S}^k \subseteq \overline{S} \text{ and } k=0} S' \cap \overline{S}^k\} \tag{2.2}$$

where $RC(\neg \overline{S} \cap S') = RC(\neg S_1 \cap \neg S_2 \ldots \cap \neg S_n \cap S')$, and

$\overline{S}^k$ denotes that the number of positively expressed sources in $\overline{S}$ is k.

A naive evaluation of this equation would require 2^n accesses, which correspond to each possible overlap of n sources with the candidate source S'. However, as we noted previously, we only maintain a small number of overlaps for areas that are deemed large. Thus, we only need to evaluate R + n overlaps, where R is the number of maintained overlaps and n is the number of sources. Nevertheless, computing the overlaps for each candidate source is computationally expensive.

The number of variables used to define the MaxEnt problem is generally much smaller than the number of maintained overlaps. Thus, we shall compute the residual coverage of a candidate source S' given that the sources in set $\overline{S}$ have been probed using the variables in the problem formation. We will maintain varList$_{\widehat{S}}$ which includes the set of variables that *do not* correspond to the set of sources in $\overline{S}$. The procedure for computing the residual coverage given the set of variables in varList$_{\widehat{S}}$ and candidate source S' is given by Procedure getResidualCoverage.

Procedure getResidualCoverage:Procedure to compute residual coverage given a set of probed sources $\overline{S}$ and a candidate source S'.

Input:
$\overline{S}$ /* Set of probed sources */
1 **S'** /* Candidate source considered */
2 **varList** /* Array of variables $\notin \overline{S}$ */
3 **varArray** /* Array of variables */
Output:
rc /* residual coverage of S' given sources in $\overline{S}$ were selected. */

```
4  var_S = generate 1-pos variable which expresses source S' positively.;
5  var_empty ← bitset of size nSources where all bits are zero;
6  rc = 0;
7  foreach i= to |varsList| do
8  |  if (varList[i] AND var_S) = var_S then
9  |  |  if (varList[i] AND var_S̄) = var_empty then
10 |  |  |  rc += varArray[varList[i]];
11 |  |  end
12 |  |  remove varList[i] from varList
13 |  end
14 end
```

Example 2.4. *Consider the set of five sources $S = \{A, B, C, D, E\}$ given in Example 2.2. Source coverage statistics and three overlap statistics are given in this example.The set of constraints and the variables generated by Procedure generateConstraints() of Algorithm 2.1 is given below.*

$P(A) = \boldsymbol{A}B'C'D'E' + \boldsymbol{AB}C'D'E' + \boldsymbol{A}B'\boldsymbol{CD}E'$

$P(B) = A'\boldsymbol{B}C'D'E' + \boldsymbol{AB}C'D'E'$

$P(C) = A'B'\boldsymbol{C}D'E' + A'B'\boldsymbol{CD}E' + \boldsymbol{A}B'\boldsymbol{CD}E'$

$P(D) = A'B'C'\boldsymbol{D}E' + A'B'\boldsymbol{CD}E' + \boldsymbol{A}B'\boldsymbol{CD}E'$

$P(E) = A'B'C'D'\boldsymbol{E}$

$P(A \cap B) = \boldsymbol{AB}C'D'E'$

$P(C \cap D) = A'B'\boldsymbol{CD}E' + \boldsymbol{A}B'\boldsymbol{CD}E'$

$P(A \cap C \cap D) = \boldsymbol{A}B'\boldsymbol{CD}E'$

Initially, varList will contain all variables created by Procedure generateConstraints(). For this example, varList is initially set to:

$\boldsymbol{varList} = \{AB'C'D'E',\ A'\boldsymbol{B}C'D'E',\ A'B'\boldsymbol{C}D'E',\ A'B'C'\boldsymbol{D}E',\ A'B'C'D'\boldsymbol{E},\ \boldsymbol{AB}C'D'E',$
$A'B'\boldsymbol{CD}E',\ \boldsymbol{A}B'\boldsymbol{CD}E',\ A'B'C'D'E'\}$

Source A is probed first followed by C, thus the set of probed sources is $\overline{S} = \{A, C\}$. *After probing sources A and C, the list of variables utilized for residual coverage computation (varList) is updated to:*

$\boldsymbol{varList} = \{\ A'\boldsymbol{B}C'D'E',\ A'B'C'\boldsymbol{D}E', A'B'C'D'\boldsymbol{E}\ \}$

The next step is to compute the residual coverage given by each unprobed source and select one with the largest residual coverage. Sources B, C, and E are considered and their residual coverage is computed w.r.t. $\overline{S}$. *Procedure residualCoverage() will consider the variables in varList and compute the sum of the variables corresponding to the candidate source* S'. *Thus, the residual coverages of sources B, C, and E are computed as follows:*

$rc(\overline{S},\ B) = A'\boldsymbol{B}C'D'E'$

$rc(\overline{S},\ D) = A'B'C'\boldsymbol{D}E'$

$rc(\overline{S},\ E) = A'B'C'D'\boldsymbol{E}$

In this example, source B has the largest residual coverage. Thus, source B is probed and the list of variables will be updated accordingly. Its obvious from this small example that the k-pos variables are much easier to deal when computing the expected residual coverage given by a source.

We present a Static Ordering strategy which invokes Algorithm 2.1 of the over-

lapEstimation component to generate overlap estimates and Procedure greedySelect() to order data sources. The Static Ordering strategy is given by Algorithm 2.2 and illustrates by the data flow diagram in Figure 2.6. Recall that the coverage and partial list of overlap

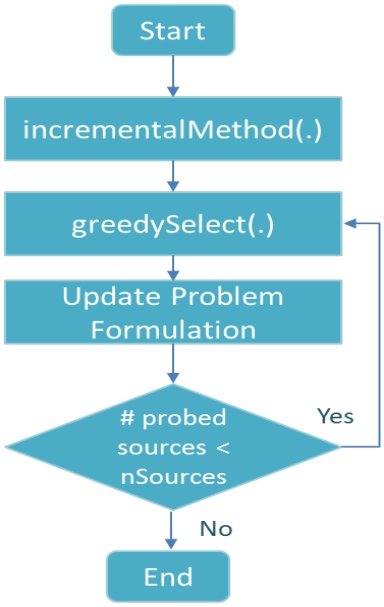


Figure 2.6: Static Ordering - Data Flow Diagram

statistic utilized by the overlapEstimation component (Algorithm 2.1) may be inaccurate or stale. For an additional small cost, accurate coverage statistic of newly probed source can be computed. Furthermore, overlap statistics can be computed between the set of probed sources and the newly probed source. These statistics can be incorporated into the MaxEnt problem formulated and the overlapEstimation (Algorithm 2.1) can be reinvoked to re-estimate the distribution of the variables under the new set of constraints. Such additional information is likely to improve the ordering of unprobed sources. Once a source S' is probed, we can obtain two pieces of information for a relatively small cost:

1. First, we can obtain the actual source coverage statistic of S', P(S').

Algorithm 2.2: Static Ordering

Input:
  $\mathbf{C}_{initial}$ */* coverage of sources */*
  $\mathbf{O}_{initial}$ */* partial overlap statistics */*
  S */* set of sources */*
  nSources */* number of sources */*
  threshold */* number of answer tuples to retrieve */*
Output:
  itemsList */* answer tuples */*
  $\prod$ */* permutation of data sources */*

```
1  ∏ = ∅;
2  O_Stats = O_initial;
3  C_Stats = C_initial;
4  varArray ← overlapEstimation(C_Stats, O_Stats);
5  S̄ ← ∅;
6  Ŝ ← S;
7  while |itemList| < threshold AND |∏| < nSources do
8  |   S' ← greedySelect(S̄, Ŝ);
9  |   itemList ← probe source S';
10 |   add S' to ∏;
11 end
```

2. Second, we can obtain the statistic describing the union of S' and the set of probed sources, $\mathrm{P}(S' \cup \overline{S})$.

3. Third, we can obtain the overlap statistics describing the intersection of S' and one or more sources from the set of probed sources.

The first two policies can be applied to compute new statistics for a fairly small cost. The third policy involves one or more statistics describing the intersection between a newly probed source and the set of probed sources. This policy will involve a large number of statistics, hence, which cannot be computed on-the-fly. In the next section, we shall revisit this problem and devise a heuristic for selecting a small subset of statistics to compute. Here, we shall concentrate on the first two policies to devise an adaptable

ordering algorithm.

Example 2.5. *Consider the set of five sources given in Example 2.2. Source coverage statistics and three overlap statistics are given in this example. Based on these statistics, Algorithm 2.2 generates the following ordering: Ordering: A C B E D*

If we consider an adaptable source ordering algorithm, one which incorporates accurate source coverage and union statistics on-the-fly, then its possible to revise the ordering of unprobed sources. For the above example, source A is probed initially, followed by source C. Once source C is probed, we can compute the overlap between sources A and C. Thus, a new statistic is added to the set of constraints: P(A ∪ C)= .6

The overlap estimation will be re-invoked to recompute the distribution, and a new source ordering can be generated. Updated Ordering: A C B D E

In this example, the next source to be probed has not changed, but last two sources in the ordering were swapped. By examining the area under the curve, we note that the new ordering is better based on the area under the curve measure. This example motivates that adaptable (or dynamic) strategy can generate a better ordering compared to a static ordering policy.

2.5 Statistic Selection

In the previous section a greedy source ordering strategy was presented. Source ordering is composed of two dimensions, one which is static while the other is adaptable, called dynamic ordering. Both ordering strategies utilize a small set of approximate

Algorithm 2.3: Dynamic Ordering

Input:
  $\mathbf{C}_{initial}$ */* coverage of sources */*
  $\mathbf{O}_{initial}$ */* partial overlap statistics */*
  S */* set of sources */*
  nSources */* number of sources */*
  threshold */* number of answer tuples to retrieve */*
Output:
  itemsList */* answer tuples */*
  $\prod$ */* permutation of data sources */*

1 $\prod = \oslash$;
2 O_Stats = $\mathrm{O}_{initial}$;
3 C_Stats = $\mathrm{C}_{initial}$;
4 $\overline{S} \leftarrow \oslash$;
5 $\widehat{S} \leftarrow$ S;
6 **while** $|itemList| < threshold\ AND\ |\prod| < nSources$ **do**
7 varArray ← overlapEstimation(C_Stats, O_Stats);
8 $S' \leftarrow$ greedySelect($\overline{S}$, $\widehat{S}$);
9 itemList ← probe source S';
10 add S' to $\prod$;
11 compute P(S');
12 compute P($S' \cup \overline{S}$);
13 update O_Stats & C_Stats;
14 **end**

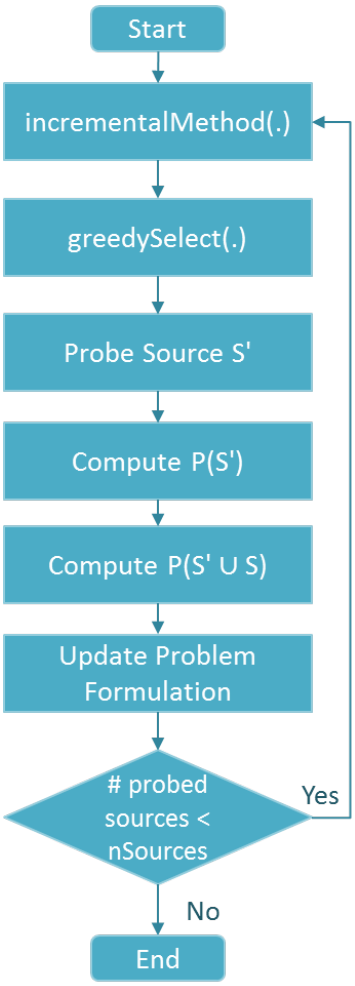


Figure 2.7: Dynamic Ordering - Data Flow Diagram

coverage and overlap statistics. This section investigates the statistic selection component which chooses the set of statistics to compute that most likely will improve the order of sources. Two ordering strategies are provided which utilize this component called static+ and dynamic+.

Consider a set of n sources S = $\{S_1, S_2, S_3, \ldots, S_n\}$. Given a source ordering $\prod$, let $\overline{S}$ denote the set of probed source in the ordering, S' denote the next source to be probed, and $\widehat{S}$ denote the unprobed sources. The challenge is to determine the set of

statistics to compute to improve the ordering of the unprobed sources. Computing all possible statistics is expensive and cannot be performed efficiently on-the-fly. Thus, the challenging problem is determining the small set of statistics that should be computed with the goal of improving the ordering of the unprobed sources. For example, consider a permutation of probed sources $\prod = \{S_{i1}S_{i2}S_{i3}\ldots S_{in}\}$ and a set of unprobed sources $\widehat{S}$ $= \{S_{k1}S_{k2}S_{k3}\ldots S_{km}\}$. Two possible statistics selections strategies can be applied: First, since the actual tuples/records of probed sources are known, all possible overlaps between probed source can be computed and is expected to improve the estimated overlaps and hence improve the ordering of unprobed sources $S_{k1}\ldots S_{km}$. Second, overlap statistics can be computed between unprobed sources. Such overlap statistics cannot be computed, but estimates can be obtained from a Third-Party statistics server. Nevertheless, in both cases, the number of statistic to consider and compute is very large an efficient and smart heuristic must be designed to determine the small subset of statistics to compute.

Through empirical studies, it was shown that overlap statistics that describe large intersection areas are more useful than overlap statistics that describe disjoint sets or a very small intersection area. Thus, one parameter that will be used to select a statistics will be the expected size of the overlap statistics and the number of sources that describe the intersection.

Since the overlap estimation problem is solved using MaxEnt. MaxEnt essentially provides the most likely distribution under the specified constraints (coverage and overlap statistics) such that entropy is maximized. Many possible distributions are possible, but only a few provide the maximum entropy. A particular solution provides the most likely distribution for the variables used to express the problem, but also provides the

sensitivity of the solution to small changes in the constraint values (coverage and overlap statistics). Sensitivity is defined as the amount of change allowed in the variable upper and lower bounds such that the value of the objective function will not change. Sensitivity essentially describes how fitted the model is, so a set of variables that are involved in several constrains (described by coverage and overlap statistics) are more likely to have very small variance (or zero variance), while variables that are involved in few constraints are more free and hence expected to have higher variance. A conjecture is made that adding additional statistics (or constraints) to the MaxEnt formulation will lower the sensitivity of the solution. Hence, its desirable to select a statistic that corresponds to variables with high sensitivity.

These two factors are used to devise 2.4 procedure. The procedure selects a set of k statistics to compute, which can correspond to probed or unprobed sources. This procedure will be utilized to introduce two new ordering strategies, static+ and dynamic+ orderings. If the statistics correspond to probed statistics, exact values of the overlap statistics can be computed since the sources have been accessed and their information processed and stored. However, if the statistics correspond to unprobed sources exact values of the overlap values cannot be obtained. Nevertheless, estimated values of the overlap statistic can be obtained via sampling or requested from a Third Party statistic server. Two ordering strategies that utilize the statisticSelection() procedure are presented, namely, Static+ (Algorithm 2.5) and Dynamic+ (Algorithm 2.6) ordering. The data flow diagram that illustrates Static+ and Dynamic+ orderings is given by Figures 2.8

Algorithm 2.4: statisticSelection()

```
   Input:
   O_Stat /* list of known overlap statistics */
   varArray /* list of variables and their estimated values */
   varArrayVariance /* list of variables and their variance */
   statCandidateList /* list of candidate overlap statistics */
   Output:
   topKList /* list of k overlap statistics */
 1 topKList ← ∅;
 2 if statCandidateList = ∅ then
 3 |  statCandidateList ← generate list of 2-way overlap statistics ∉ O_Stat;
 4 end
 5 foreach statistic S ∈ statCandidateList do
 6 |  if expected size of S AND variance of S ≤ threshold then
 7 |  |  remove S from statCandidateList;
 8 |  |  list ← generate list of (k+1)-way statistics corresponding to S ∉
   |  |  O_Stat;
 9 |  |  statCandidateList ← list;
10 |  else
11 |  |  if expected size of S AND variance of S > min in topKList then
12 |  |  |  topKList ← add S;
13 |  |  end
14 |  end
15 end
```

and 2.9. The presented ordering strategies can be viewed as having two dimensions. The first dimension controls the adaptability of the orderings, thus from this we derived two adaptable algorithms Dynamic and Dynamic+. The second dimension controls whether *additional* statistics are incorporated. From this we derive two orderings, Static+ and Dynamic+. The summary of the orderings is illustrated by Figure 2.10.

2.6 Related Work

Previous work has examined query answering and optimization in data integration systems where overlap exists between sources. Our work differs from previous work in

Algorithm 2.5: Static+

Input:
C$_{initial}$ */* coverage of sources */*
O$_{initial}$ */* partial overlap statistics */*
S */* set of sources */*
nSources */* number of sources */*
threshold */* number of answer tuples to retrieve */*
Output:
itemsList */* answer tuples */*
$\prod$ */* permutation of data sources */*

```
1  O_Stats = O_initial;
2  C_Stats = C_initial;
3  O_Stats ← statisticsSelection();
4  varArray ← overlapEstimation(C_Stats, O_Stats);
5  S̄ ← ∅;
6  Ŝ ← S;
7  while |itemList| < threshold AND |∏| < nSources do
8  |   S' ← greedySelect(S̄, Ŝ);
9  |   itemList ← probe source S';
10 |   add S' to ∏;
11 end
```

Algorithm 2.6: Dynamic+

Input:
C$_{initial}$ */* coverage of sources */*
O$_{initial}$ */* partial overlap statistics */*
S */* set of sources */*
nSources */* number of sources */*
threshold */* number of answer tuples to retrieve */*
Output:
itemsList */* answer tuples */*
$\prod$ */* permutation of data sources */*

```
1  nItems = 0;
2  O_Stats = O_initial;
3  C_Stats = C_initial;
4  while —itemList— < threshold AND |∏| < nSources do
5  |   varArray ← overlapEstimation(C_Stats, O_Stats);
6  |   invoke Thread 1 and Thread 2;
7  |   update O_Stats and C_Stats;
8  end
```

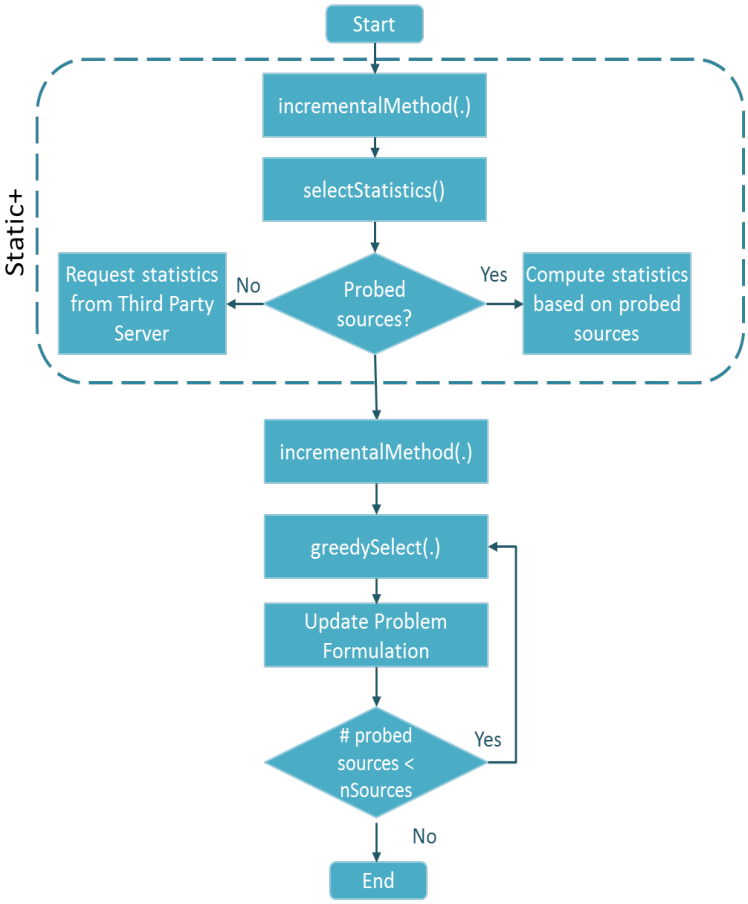


Figure 2.8: Static+ Ordering - Data Flow Diagram

two ways. First, we presented an on-line and scalable method to generate source overlap estimates. Second, we presented a dynamic source ordering strategy, that incorporates new statisitical information, re-generates overlap estimates, and adapts source ordering in a on-line fashion.

In [29], probabilistic information about overlap between sources is used to help in choosing the k most useful sources to access. This paper used information about domain overlap, i.e, overlap between the collections of objects in the schema, because of the exponential blowup when using source overlap information. This work was later

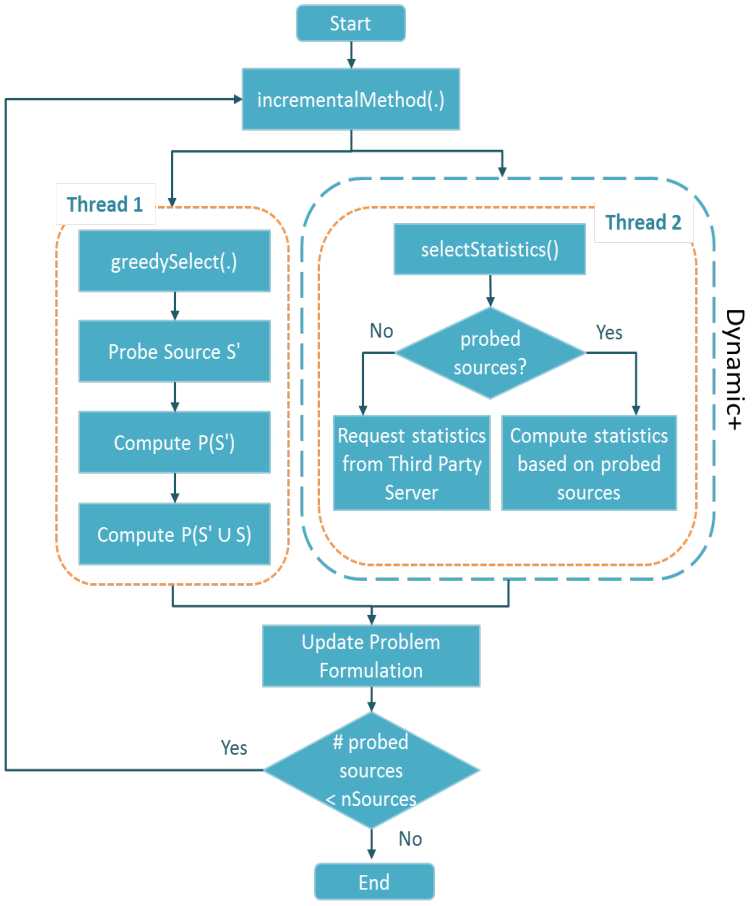


Figure 2.9: Dyanmic+ Ordering - Data Flow Diagram

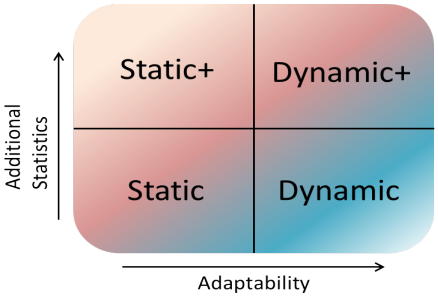


Figure 2.10: 2-Dimensions to Source Ordering

extended and appeared in [38] as the Tukwila Data Integration System.

Using knowledge of source overlaps for query optimization was discussed in [80]. Two challenges were outlined in using source overlap information to optimize query plans.

The first challenge is the amount of information required to compute source overlaps is exponential in the number of sources. Second, the naive algorithm which uses the overlap information to choose the best set of sources to provide full or partial answers is also exponential in the number of sources. Sampling was considered a good option to derive partial statics describing source overlaps, which in turn can be used to compute approximation of other overlaps using maximum likelihood estimators. This paper, however, did not address which sampling should be

In [55], the problem of obtaining a complete result set with minimum cost in data integration systems is considered. Each data source is associated with a extensional value (number of tuples a source contains) and intentional value (number of attributes a source contains). These two measures are used to determine the 'completeness' of the answer set provided by a given source, and is utilized to determine the set of data sources to query to retrieved the complete set of query answers with minimum cost. This paper, however, did not consider overlap between data sources.

In [11], a data integration system for life science data is considered where there is an overlap of objects in the sources. Query answering requires the traversal of the alternative paths between sources to obtain the result object set. Each path is associate with a benefit score and cost for traveling that path. Several greedy algorithms are presented for choosing the set of paths based on the benefit and cost of each traversing each path. This paper assumes that the target set of objects, and the overlapping objects in the data sources are known beforehand. In [65], the problem was revisited and the authors discussed techniques for estimating overlapping objects between different sources.

The authors in [57, 58, 56] presented StatMiner. The StatMiner system [57,

56]learn coverage and overlap statistics for a large set of data sources by learning association rules for query classes. A multi-objective query processor for data integration system is presented which uses the coverage and overlap statistics learned by StatMiner, as well as, cost of accessing and querying data sources to determine the best query plan. Coverage and overlap statistics are learned for large query classes. To control the number of statistics learned and stored, the StatMiner system maintains those statics for large query classes which are above specified threshold. An adaptive algorithm is applied to determine the level of resolution of the learned statistics. Two greedy algorithms are presented, Simple Greedy and Greedy Select, to select the subset of data sources which provide a complete answer set.

In [23], the authors present ROSCO a system for query answering for text collections. Sampling is used to learn the coverage, relevance, and overlap statistics for query classes, which are defined in terms of frequent keyword sets. An off line approach is presented which stores overlap statistics collected by ROSCO with respect to query classes. Data mining is applied on the text collections off line to compute frequent items, and then statistics are computed for the different sets. Statistics are maintained for large sets only. When a query arrives to the system, it is mapped to a set of items and the precomputed statistics are received. An online approach is also presented which computes overlap statistics by sampling each text collection at run time. Experimental results showed that the online approach performed better when answering top-k queries, where k is rather small compared to the number of relevance sources in the collection.

Query answering in the presence of overlapping data has been considered for a peer data management system (PDMS). In [69, 68, 70] the links between neighboring

peers to determine whether there is any benefit to send a query to that neighbor.

2.7 Experimental Results

This section presents an experimental study of our on-line query answering system. We show that (1) our system can generate overlap estimates between sources, (2) utilize overlap estimates for source ordering to return answers quickly.

2.7.1 Data Set:

The dataset used is a snapshot of computer science books listing from AbeBooks.com. The data collection includes 1000 bookstores (sources) and 25347 book records (only 1256 books). The cost model depends on two factors, access overhead cost and the per-tuple cost. The mean connection time to a website listed on AbeBooks.com is approximately 756 msec. The per-tuple retrieval cost is approximately 0.3 msec. These costs will be used as the standard access overhead cost and per-tuple retrieval cost for all data sources.

We assume a Statistic Server exists which can provide *approximate* statistics about any subset of sources for a small overhead cost. The overhead for obtaining approximate statistics will be set to a constant value of 250 msec. This overhead represents the time to send the request to the server and receive the answer. This component is not strictly tied with our on-line query answering system. If such a sever does exist, then the query answering system can restrict new statistics to probed sources only or generate overlap statistics between unprobed sources based on sampling.

2.7.2 Experimental Setup

We implemented the following ordering algorithms:

- ***Random:*** probes sources in a random order.
- ***Naive:*** probes sources using greedySelect(.) algorithm with utilizing only source coverage information.
- ***Static:*** probes sources using greedySelect(.) algorithm with k overlap statistics as input and utilizes incrementalMethod() algorithm to estimate source overlaps.
- ***Static+:*** similar to Static ordering, but also utilizes k additional overlap statistics which are incorporated into the problem formulation early on.
- ***Dynamic:*** probes sources using greedySelect(.) algorithm with partial overlap statistics as input and utilizes incremetnalMethod() algorithm to estimate source overlaps.
- ***Dynamic+:*** similar to Dynamic ordering, but also utilizes k additional overlap statistics as input and utilizes incremetnalMethod() algorithm to estimate source overlaps, and repeatedly applies source probing techniques on probed sources.

All algorithms were implemented in Java 1.5, and experiments were run under Eclipse on a Windows 7 machine with 2.40GHz Intel Core i3 CPU and 4GB of RAM. The algorithms utilized the LINDO solver for solving the MaxEnt problem.

2.7.3 Experimental Evaluation

2.7.3.1 Evaluating Overlap Estimation Parameters

The Overlap Estimation Component utilized several parameters to control the size of the problem and accuracy of the estimated overlaps. Procedure expandConstraints() utilized nMaxExpanded parameter, which specified the maximum number of variables to create (or add to the problem formulation) at a given round. An increase in this parameter results in a large problem formulation, and thus longer processing time. However, since more variables are being specified, the accuracy of the estimated overlaps is expected to improve as this parameter is increased. Figure 2.11 provides the relative error between true and estimated overlaps (via the Overlap Estimation Component) for varying number of initial overlap statistics. As expected, the relative error decreases with increasing number of statistics since additional statistics allows for more accurate estimation. Note, however, that the processing time increases as well with the addition of new statistics, since additional statistics implies that the problem is defined with more variables which require a longer processing time. Figure 2.12 illustrates the effect of varying nMaxExpanded on the relative error and total processing time. Based on these experiments, a value of 200 is considered good for the nMaxExpanded parameter.

2.7.3.2 Evaluating Ordering Strategies

In Figure 2.13 we compare the performance of Random, Naive, Static, Static+, Dynamic, and Dynamic+ orderings for obtaining 90% coverage. In Figure 2.13(a) the

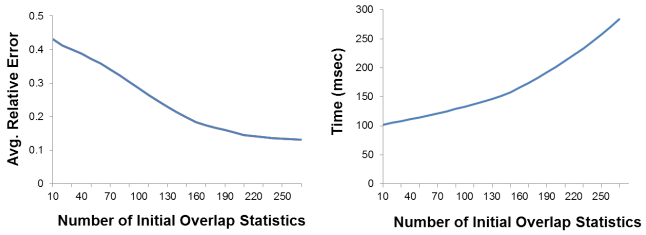


Figure 2.11: Plot of (a) relative error and (b) total processing time for Overlap Estimation Component while varying number of initial overlap statistics

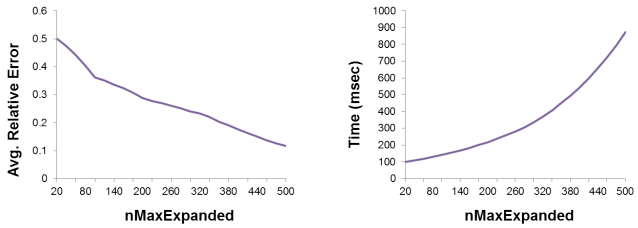
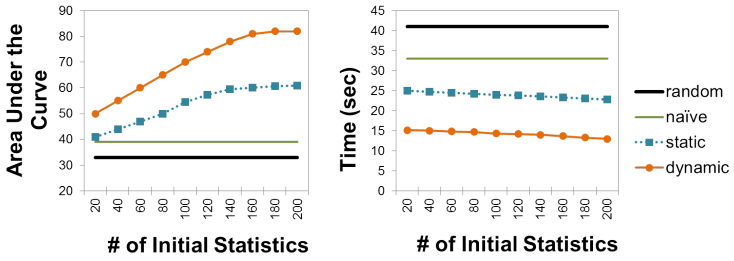


Figure 2.12: Plot of (a) relative error and (b) total processing time for Overlap Estimation Component while varying nMaxExpanded



(a) Area Under the Curve (b) Processing time (sec)

Figure 2.13: Plot of (a) area under the curve and (b) total processing time to obtain 90% total coverage.

area under the curve is plotted for varying number of inital statisitsc provided to Static, Static+, Dynamic and Dynamic+, while total processing time is shown in Figure 2.13(b). The same set of random statisitics was provided to all orderings. As the number of inital statisitics is increased, the measured area under the curve also increases thus indicating

a better ordering. However, the improvement observed in the area is minimal when more statisitics are provided. This shows that a large number of statisitics is not necessarly helpful. Figure 2.13(b) plots the total processing time as the number of inital statisitics is increased. As the number of inital statistics are increased, the processing time is expected to increase since a large set of statitistics will increase the processing time of the Overlap Estimation component. However, as we observe in Figure 2.13(b) a large set of statisitics helps ordering strategies to select a better source ordering and hence a smaller set of sources to obtain 90% coverage. Observe that both Random and Naive orderings are constant for varying number of input statitiscs because these two ordering do not make use of the overlap statistics.

Both Dynamic+ and Static+ orderings compute and/or request a total of n additional statistics. Next, we shall evaluate the effect of the various parameters and statistic selection strategies utilized by Dynamic+ and Static+. Table 2.3 lists the parameters and the range of values considered. The initial set of overlap statistics provided to both orderings included approximate estimates of source coverages and 20 randomly selected 2-way overlap statistics. Figures 2.14 shows the area under the curve for Dynamic+ and Static+ orderings for varying values of n. The other parameters, which include kMax, StatisticPolicy, and SelectionPolicy were set to 2, unresistricted, and all, respectively. As the value of n is increased, Dynamic+ and Static+ orderings are able to generate better source ordering as captured by the area under the curve in Figure 2.14. However, it should be noted that with increasing values of n, the problem size will increase and will require more time to compute.

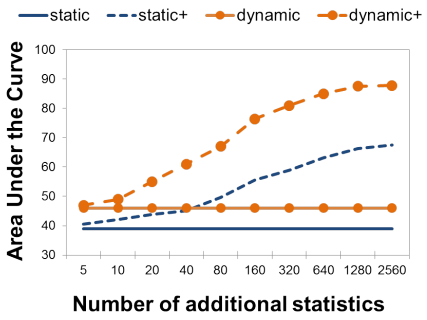


Figure 2.14: Plot of area under the curve for varying number of n (# of additional statistics)

Table 2.3: Dynamic+ ordering parameters

n	(10 - 100) Total number of statistics to request/compute
kMax	(2 - 5) Defines max k-way statistics to evaluate and request/compute
StatisticPolicy	0 - compute overlap between probed sources, 1 - request overlap involving un-probed sources, 2 - unresitricted
SelectionStrategy	0 - variance, 1 - expected overlap size, 2 - diversity, 3 - all.

Next, we evaluate the performance for varying statistic selection policies. In Figure 2.15 the parameter StatisticPolicy is set to either 0, 1,or 2, which specifies whether selected statistic should be redistricted to probed sources, unprobed sources, or both. Parameters n, kMax, and SelectionStrategy are set to 120, 2, and all, respectively. Parameter SelectionStrategy specifies the characteristics considered for selecting statistics. In Figure 2.15 the strategy used for selecting statistics to compute or request from a Third Party server is varied to consider statistics with higher variance, expected overlap size, diversity, or all three.

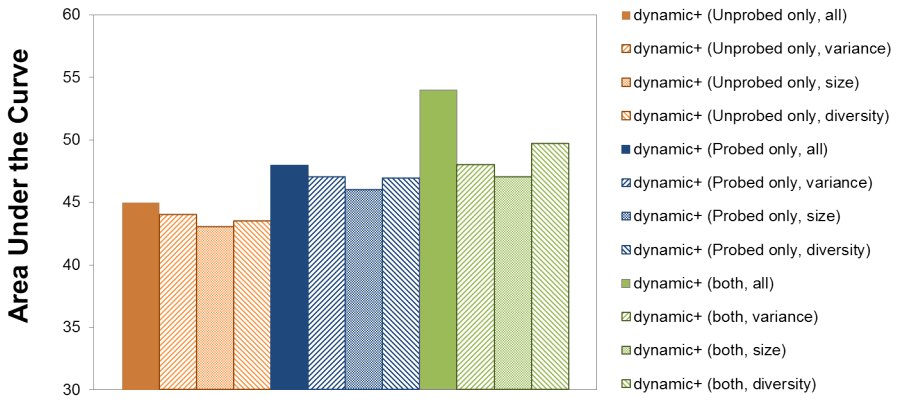


Figure 2.15: Plot of area under the curve for varying values of parameters StatisticPolicy and SelectionStrategy.

2.8 Final Remarks

This chapter presented an Online Answering System for Integrated Sources (OASIS). OASIS is the first online, adaptable, query answering systems which considers source coverage, overlap between data sources, and source access cost in ordering source accesses. The first component of the system is the overlap estimation, which is a fast and scalable method to generate overlap estimates between data sources. The second component of the system is source ordering which generates an ordering of source accesses using the overlap estimates generated by the overlap estimation component. The last component of the system selects a set of statistics to compute to improve the ordering of sources. Future work includes joining our techniques with those that consider quality measures in query answering, such as data freshness, truth discovery, and data copying.

Chapter 3

Optimizing Candidate Document Selection for Query Answering

3.1 Introduction

Extensible Markup Language (XML) is being increasingly adopted as a data format due to its extensible, portable and self-describing nature. With the growth of XML-encoded information there is an increased need for effective query answering on these XML repositories. Generally, *query answering* is composed of several phases: XPath/XQuery parsing, query rewriting and optimization, retrieving candidate documents, and performing query matching. Since the database contains a large set of documents, index structures are usually employed for effective retrieval of candidate documents. Nevertheless, the retrieval phase may return a large set of candidate documents, and since the cost of processing each candidate document varies (based on document length or query characteristics), the order in which documents must be processed should be considered to optimize

the response time. In this chapter, two optimization problems are considered, optimal ordering and selection of candidate documents, which provide complementary approaches to dealing with a large number of candidate documents. First, the the *document ordering* problem is examined, which finds the best ordering of candidate documents such that the expected time to the first k matched documents is minimized. Second, the *document selection* problem is considered, which identifies a subset of candidate documents such that the expected number of matched documents is maximized and the total processing time does not exceed a given upper bound.

These two optimization problems are considered for applications where document precedence constraints exist which restrict the order in which documents must be processed. Such document precedence constraints can be modeled as a set of chains, and documents in a given chain must be processed in the order specified. In this context, directed links exist from one document to another that either describe precedence in terms of versions, access patterns or complementary relationships. This problem is important in several applications. Consider a repository that stores Blog postings. When a new Blog post is added to the repository, a link is created between this new Blog posting and any related Blogs (threads) that exist in the repository. This structure will form a linked chain of documents (or more complex graphs). When such precedence constraints describe complementary relationships between documents, thus, this information can be leveraged by placing preference to the retrieval of a set of chained documents versus un-chained documents.

Document relevance based on the Vector Space Model is usually utilized for document ranking. Previous work has been presented on top-k queries [49, 85, 19, 21, 51]

and document ranking for Information Retrieval [79, 12, 48]. Part of the work on top-k queries focused on keyword searches [49, 85, 19, 21], while other work presented top-k algorithms for approximate or relaxed query processing [51].

The major focus of past work has been on ranking of query results based on relevance. This chapter investigates candidate document ranking based on two parameters, the probability of query matching and document processing time. The objective of ranking is to minimize the response or maximize the number of matches for a given time bound.

The contributions of this chapter are summarized as follows:

- An estimation model is presented for computing the probability of a given query having a match in a document and the expected processing time of a given document based on its features, such as size, average depth, number of recursive elements, etc. These two parameters are used to formulate the objective functions presented for the ordering and selection problems.

- Polynomial time algorithms are presented for the document ordering problem for applications with or without precedence constraints. A proof of the optimality of these algorithms are presented when considering the first matched document. Experimental results show that these algorithms also minimize the expected time to the first k matches, where k is a small number compared to the total number of matches.

- Since the selection problem is NP-complete, a heuristic algorithm is presented and the experimental results have shown that this algorithm is effective.

The rest of the chapter is outlined as follows: In Section 3.2, a description of how to compute the probability of query match and the document processing time is provided. In Section 3.3 the optimal document ordering problem is presented, while in Section 3.4 optimal document selection problem. In Section 3.5 experimental results are presented, and conclusions appear in Section 3.6.

3.2 Preliminaries

In this section, the models utilized to estimate the two parameters, the probability of a query match and the document processing time, used to express the objective functions for the two optimization problems are discussed.

3.2.1 Estimating the Probability of Query Match

To compute the probability of a query having a match in a candidate document, the Vector Space Model for Information Retrieval and tf-idf are utlized to compute the relevance of the document with respect to a set of query terms. Traditionally, in flat-documents, *terms* refer to keywords or values, but in the XML-context, terms are expressed as elements, values, parent-child pairs, or paths in the XML document or XPath query.

For efficient retrieval of candidate documents, XML databases, such as MarkLogic [2], usually leverage an inverted index to retrieve the set of candidate documents for a given query. The inverted index is built over the collection of documents, such that for every distinct term t in the collection a list of document identifiers (d_i) is stored. To

retrieve candidate documents for a given query, the query is first decomposed into a set of terms which allows probing the index for the list of candidate documents. A given document is retrieved if all query terms occur in the document. The inverted index is augmented to contain the frequency of a given term in the documents. Each entry in the index is a 2-tuple (d_j,tf_{ij}) , where d_j denotes the document identifier and tf_{ij} denotes the term frequency of t_i in d_j.

The term frequency(tf) is defined as follows: $tf_{ij} = \frac{n_{ij}}{\sum_{t_k \in d_j} n_{kj}}$

- n_{ij} = occurrences of term t_i in document d_j
- $\sum_{t_k \in d_j} n_{kj}$= sum of the occurrences of all terms in d_j

The inverse document frequency (idf) is a measure of the general importance of the term and is defined as follows: $idf_i = \log \frac{|D|}{1+|t_i \in d|}$

- $|D|$ = total number of documents
- $|t_i \in d|$ = number of documents in which t_i occurs

The statistics used to compute tf-idf are computed off-line when a document is first inserted into the database. The term-frequency (tf_{ij}) for each term t_i and document d_j is pre-computed and stored in the augmented index. The inverse document frequency (idf_i) is computed for each term t_i over the entire collection of documents and is stored in an array which can be accessed by the term identifier.

Given a query q, the query is decomposed into a set of terms T = (t_1, t_2, ..., t_k), which denote elements, values, parent-child pairs, or paths in the query. For each candidate document, the (tf_{ij} x idf_i) score is computed for each term in the set T. The

overall query relevance score for a given document is defined as follows:

$$Score(q|d_j) = \sum_{i=1}^{k} (tf_{ij} \times idf_i) \tag{3.1}$$

The above score provides the relevance of a given document for a specified query. To determine the probability of a query having a match in a given document, a model based on logistic regression is adopted to estimate the conditional probability of a query having a match in a given document using the computed relevance scores[6]. The logistic model is fitted with training data and computed off-line, and a one-to-one correspondence is created between the relevance scores and the probabilities.

3.2.2 Document Processing Time

The other parameter considered is the processing time of a given candidate XML document. The processing time depends on many factors, such as (1) the query matching algorithm, (2) availability of an index on the documents, (3) complexity of the query, and (4) features of the documents, such as document size, average document depth, number and level of recursive elements in the document, etc. The document processing time is computed based on the features of the document since the other factors equally affect the processing time of each candidate document. The document features considered are: (1) the size of the XML document, (2) recursion level, and (3) average path depth.

The size of the XML document is the number of element and value nodes in the XML tree. The recursion level of a node in the XML tree is defined as the maximum occurrence of the node along any root-to-leaf path in the tree. The recursion level of an

XML document is defined to be the maximum recursion level of any node in the tree. The third feature is the average depth of all paths in the XML tree. These three features are proportional to the processing time and are used to estimate the processing time, which is then stored in an array that can be accessed via the document identifier.

3.3 Optimal Document Ordering

This section addresses the document ordering problem to minimize the response time, which is defined as the expected time to the first matched document, expected time to the second matched document (after the first matched document is found), and so forth. The optimal ordering problem has been considered in job scheduling but with an objective function and parameters that are not applicable to the ordering and selection problems considered in this dissertation.

An objective function is presented which expresses the expected response time using the two parameters defined in Section 3.2. It assume that the probability of a query matching a given document is independent of its probability of having a match in another document. The objective function computes the response time to next matched document, given that directly consider the number of matches in each document. However, it is noted that the number of matches in a document is proportional to the probability of a document having a match. A polynomial time algorithm is presented for generating an optimal document permutation for applications with and without precedence constraints.

Given a set of documents D = $\{d_1, d_2, d_3, \ldots, d_n\}$, let $\prod$ denote an ordering (or permutation) of the documents in set D. The expected time to the first matched document

is given by the following objective function:

$$\begin{aligned} E(\prod) &= T_{\prod(1)} + (1 - P_{\prod(1)})T_{\prod(2)} + \\ (1 - P_{\prod(1)})(1 - P_{\prod(2)})T_{\prod(3)} + \ldots &+ (1 - P_{\prod(1)}) \ldots (1 - P_{\prod(n-1)})T_{\prod(n)} \end{aligned} \quad (3.2)$$

where, $\prod(i)$ denotes the ith document in the permutation,

$\mathbf{T}_i$ is the expected time to process document d$_i$,

$\mathbf{P}_i$ is the probability of finding a match in document d$_i$

An optimal permutation of documents, denoted by $\prod$, is a sequencing of the documents such that the time to first matched document is minimized. Such an optimal ordering can be obtained by Algorithm 3.1 in O(n log n). Theorem 3.1 provides a proof of the optimality of Algorithm 1. Furthermore, experimental results have shown that Algorithm 3.1 is an effective heuristic algorithm to minimize the time to the first k matched documents, where k is a small number compared to the total number of matched documents.

Algorithm 3.1: Optimal Document Ordering

Input: candidateList *// retrieved list of candidate documents for a given query*
Output: Output: $\prod$
1 **foreach** *document d_i in candidateList* **do**
2 | compute ratio $\frac{T_i}{P_i}$
3 **end**
4 $\prod \leftarrow$ sort documents in ascending order of their ratios;

Theorem 3.1. *Consider a set of documents D= $\{d_1, d_2 \ldots d_n\}$. A permutation of the documents, $\prod$, is an optimal permutation with respect to the objective function (3.2) iff*

$\frac{T_{\prod(i)}}{P_{\prod(i)}} \leq \frac{T_{\prod(i+1)}}{P_{\prod(i+1)}}$, *where 1 ≤ i ≤ n-1.*

Proof. Let $\prod$ be a permutation of the documents in D. Let $\prod'$ be another permutation obtained from $\prod$ by interchanging two adjacent documents $\prod(i)$ and $\prod(i+1)$, where 1 ≤ i ≤ n-1. First, it will be shown that $E(\prod) - E(\prod') \leq 0$ iff $\frac{T_{\prod(i)}}{P_{\prod(i)}} \leq \frac{T_{\prod(i+1)}}{P_{\prod(i+1)}}$. Using the objective function in (3.2),

$$E(\prod) - E(\prod') = A \times T_{\prod(i)} + A \times (1 - P_{\prod(i)}) T_{\prod(i+1)} - A \times T_{\prod(i+1)}$$

$$- A \times (1 - P_{\prod(i+1)}) \times T_{\prod(i)}$$

$$where\ A = (1 - P_{\prod(1)}) \times (1 - P_{\prod(2)}) \ldots (1 - P_{\prod(i-1)})$$

$$E(\prod) - E(\prod') = A \times T_{\prod(i)} \times P_{\prod(i+1)} - A \times T_{\prod(i+1)} \times P_{\prod(i)}$$

$$Thus, E(\prod) - E(\prod') \leq 0\ iff\ \frac{T_{\prod(i)}}{P_{\prod(i)}} \leq \frac{T_{\prod(i+1)}}{P_{\prod(i+1)}}. \tag{3.3}$$

A necessary and sufficent conditions will be shown to be true to prove the theorem.

Necessary Condition: Let $\prod$ be an optimal permutation, and let $\prod'$ be a permutation obtained from $\prod$ by interchanging two adjacent documents $\prod(i)$ and $\prod(i+1)$. Since $\prod$ is optimal, $E(\prod) - E(\prod') \leq 0$. Therefore, by statement (3.3) $\frac{T_{\prod(i)}}{P_{\prod(i)}} \leq \frac{T_{\prod(i+1)}}{P_{\prod(i+1)}}$ for 1≤i≤n-1. **Sufficient Condition:** Let $\prod$ be a permutation which satisfies the inequality $\frac{T_{\prod(i)}}{P_{\prod(i)}} \leq \frac{T_{\prod(i+1)}}{P_{\prod(i+1)}}$ for 1≤i≤n-1. The permutation $\prod$ will be shown to be optimal. Let $\prod'$ be an optimal permutation, thus, $\prod'$ satisfies the inequality $\frac{T_{\prod'(i)}}{P_{\prod'(i)}} \leq \frac{T_{\prod'(i+1)}}{P_{\prod'(i+1)}}$ for 1≤i≤n-1. Since, both $\prod$ and $\prod'$ satisfy the inequality, either the two permutations are identical or one can be obtained from the other by interchanging adjacent documents with equal ratios. By statement (3.3) interchanging adjacent documents with equal ratios does

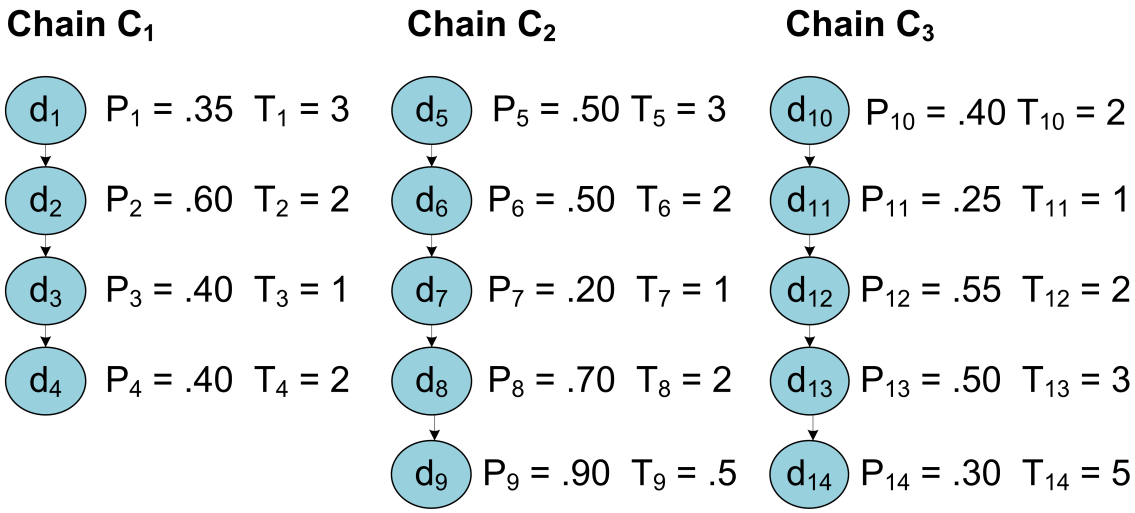


Figure 3.1: Example of documents with precedence constraints modeled by chains.

not change the value of the objective function in (3.2), implying $E(\prod) = E(\prod')$. Thus, it is concluded that permutation $\prod$ must be optimal. □

Next, the problem of optimal document ordering with precedence constraints is considered. Precedence constraints are modeled as linear chains. In a given chain, directed links exist from one document to another that either describe precedence in terms of versions, access patterns or complementary relationships. The problem of document ordering must obey these relationships when generating an optimal document permutation. Consider a set of document chains shown in Figure 3.1. The links between documents denote the precedence order that must be obeyed during query processing. For example, consider chain C_1, d_1 must be processed before d_2, similarly, d_2 must be processed before d_3 and so forth. Two ordering policies are considered for documents with precedence constraints, chain ordering and interleaved sub-chain ordering. The general chain ordering policy sets the restriction that the documents in a given chain must be processed as a

whole and in an order that maintains precedence constraints. The interleaved sub-chain ordering policy still enforces the precedence constraints, but allows document segments (or sub-chains) from different chains to be interleaved. To present algorithms for both chain ordering and interleaved sub-chain ordering policies, a formal definition of chain and sub-chain sequence ratio is given by Definition 3.1.

Definition 3.1 (Sequence Ratio)**.** *Without loss of generality, let S be a sequence of documents, such that* $S = d_1, d_2, \ldots, d_k$*. The sequence ratio of S, denoted by Ratio*(S)*, is formally defined as:*

$$\frac{T_1 + (1 - P_1)T_2 + \ldots + (1 - P_1)(1 - P_2)\ldots(1 - P_{k-1})T_k}{1 - (1 - P_1)(1 - P_2)\ldots(1 - P_k)} \tag{3.4}$$

, where the numerator is the expected processing time of the sequence, and the denominator is the probability of finding a match in any document in the sequence.

The general chain ordering policy computes the chain ratio for each chain, the chains are then ordered in ascending order of their ratio, and the documents are processed in accordance with the chain ordering (see Algorithm 3.2). The chain ratios can be computed in linear time and the sorting requires O(n log n), thus the complexity of Algorithm 2 is O(n log n).

Example 3.1. *Consider the set of chains shown in Figure 3.1. The Ratio(*C_1*), Ratio(*C_2*), and Ratio(*C_3*) are 5.45, 5.55, and 4.49, respectively. Since* $C_3 \leq C_1 \leq C_2$*, the documents in chain* C_3 *must be processed first, followed by the documents in chain* C_1*, and then documents in chain* C_2*. The document ordering generated for this example based on the*

Algorithm 3.2: :Optimal Document Ordering Under Precedence Constraints using General Chain Ordering Policy

Input: C_1, C_2, C_3 ... C_k // *chains of documents*
Output: $\prod$ // *optimal document permutation*
foreach *chain* C_i **do**
  Ratio(C_i) ← compute ratio per Definition 3.1
end
$\prod$ ← sort chains in ascending order of their ratio

general chain ordering policy is d_{10}, d_{11}, d_{12}, d_{13}, d_{14}, d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9.

The following lemma is needed to prove the optimality of Algorithm 3.2.

Lemma 3.1. *Given a set of documents* $D = \{d_1, d_2, \ldots, d_n\}$ *with precedence constraints. Let* $\prod$ *and* $\prod'$ *be two permutations of the documents in D, such that* $\prod'$ *can be obtained from* $\prod$ *by interchanging two sequences,* $\prod(i \ldots j-1) = \prod(i)\prod(i+1)\ldots\prod(j-1)$ *and* $\prod(j \ldots k) = \prod(j)\prod(j+1)\ldots\prod(k)$ *without violating the precedence constraints, where 1* $\leq i \leq j \leq k \leq n$. *Therefore, E(*$\prod$*) - (*$\prod'$*)* $\leq$ *0 iff R(*$\prod(i \ldots j-1)$*)* $\leq$ *R(*$\prod(j \ldots k)$*).*

Proof. For any given sequence of documents S = d_1 ... d_k, E(S) and P(S) are defined as follows:

$E(S) = T_1 + (1-P_1)T_2 \ldots + (1-P_1)\ldots(1-P_{k-1})T_k$ and $P(S) = (1-P_1)(1-P_2)\ldots(1-P_k)$

Using this notation, the sequence ratio can be written as: Ratio(S) = $\frac{E(S)}{1-P(S)}$

Let $\prod$ and $\prod'$ be two permutations of the documents in D, such that $\prod'$ can be obtained from $\prod$ by interchanging two sequences, $\prod(i \ldots j-1)$ and $\prod(j \ldots k)$ without violating the precedence constraints, where 1 $\leq$ i $\leq$ j $\leq$ k$\leq$ n. Using the objective function in (3.2),

$E(\prod) - E(\prod') = A \times E(\prod(i \ldots j-1)) + A \times (1 - P(\prod(i \ldots j-1))) \times E(\prod(j \ldots k)) -$

$A \times E(\prod(j \ldots k)) - A \times (1 - P(\prod(j \ldots k))) \times E(\prod(i \ldots j-1))$,

where $A = (1 - P(\prod(1))) \times (1 - P(\prod(2))) \ldots (1 - P(\prod(i-1)))$.

Simplifying the expression it is deduced that:

$E(\prod) - E(\prod') =$

$A \times E(\prod(i \ldots j-1)) \times P(\prod(j \ldots k))$ - $A \times E(\prod(j \ldots k)) \times P(\prod(i \ldots j-1))$

Thus,

$$E(\prod) - E(\prod') \leq 0 \qquad iff$$
$$\frac{E(\prod(i \ldots j-1))}{1 - P(\prod(i \ldots j-1))} \leq \frac{E(\prod(j \ldots k)}{1 - P(\prod(j \ldots k))} \tag{3.5}$$

Thus, $E(\prod) - E(\prod') \leq 0$ iff $R(\prod(i \ldots j-1)) \leq R(\prod(j \ldots k))$ It is concluded that the lemma is true. □

Theorem 3.2. *Algorithm 3.2 provides an optimal ordering of the documents in O(n log n), where their precedence constraints are described as chains.*

Proof. Let $\prod$ be a permutation such that $\text{Ratio}(\text{C}_i) \leq \text{Ratio}(\text{C}_j) \ldots \leq \text{Ratio}(\text{C}_k)$, where C_i, $\text{C}_j \ldots \text{C}_k$ are chains of documents. Permutation $\prod$ will be shown to be optimal. Let $\prod'$ be an optimal permutation, thus, $\prod'$ satisfies the inequality of Lemma 3.2. Since, both $\prod$ and $\prod'$ satisfy the inequality, either the two permutations are identical or one can be obtained from the other by exchanging adjacent chains with equal ratios. By Lemma 3.1, exchanging adjacent segments (or whole chains) of documents with equal ratios does not affect the value of the objective function in (2). Thus, it is concluded that $\prod$ must be optimal. □

Next, the interleaved chain ordering policy is presented. This policy constructs an optimal document ordering by interleaving sub-chains (or chain segments), where a sub-chain is defined as follows:

Definition 3.2 (Sub-Chain)**.** *Let C_i be a chain of documents. Without loss of generality, assume $C_i = d_1, d_2, \ldots d_n$. A subchain $C_{i,j}$ of C_i is any sequence of documents $d_1 \ldots d_j$, where $1 \leq j \leq |C_i|$.*

Algorithm 3.3 provides the details for interleaved sub-chain ordering. The sub-chain ratio of $C_{i,j+1}$ ($R(C_{i,j+1})$) can be computed in constant time from the numerator and denominator of sub-chain ratio $C_{i,j}$ ($R(C_{i,j})$). Thus, the computation of all sub-chain ratios for a given chain is linearly proportional to the number of documents in the chain. Thus, the computations of sub-chain ratios for all chains is O(n). The sub-chain ratios must be recomputed once a sub-chain is selected and this can be performed a maximum x of n times, thus, the complexity of the interleaved sub-chain ordering algorithm is $O(n^2)$ by maintaining the numerator and denominator of all sub-chain ratios.

Algorithm 3.3: : Optimal Document Ordering Under Precedence Constraints using Interleaved Chain Ordering Policy

Input:
$C_1, C_2, C_3 \ldots C_n$ */* document chains */*
k */* number of documents */*
Output: $\prod$ */* optimal permutation */*

```
1 foreach chain C_i do
2 |  compute the ratios of sub-chains in C_i per Definition 3.1
3 end
4 while |∏| != k do
5 |  let C_{i,m} be a sub-chain with minimum ratio;
6 |  add sub-chain C_{i,m} to ∏;
7 |  remove docs of sub-chain C_{i,m} from chain C_i;
8 |  recompute the ratios of the sub-chains in C_i;
9 end
```

Example 3.2. Consider the example shown in Figure 3.1 which has three chains that describe the document precedence constraints. The sub-chain ratios are computed as follows:

$ChainC_1$	$ChainC_2$	$ChainC_3$
$Ratio(C_{1,1}) = 8.57$	$Ratio(C_{2,5}) = 6.00$	$Ratio(C_{3,10}) = 5.00$
$Ratio(C_{1,2}) = 5.81$	$Ratio(C_{2,6}) = 5.33$	$Ratio(C_{3,11}) = 4.73$
$Ratio(C_{1,3}) = 5.49$	$Ratio(C_{2,7}) = 6.25$	$Ratio(C_{3,12}) = 4.39$
$Ratio(C_{1,4}) = 5.45$	$Ratio(C_{2,8}) = 5.74$	$Ratio(C_{3,13}) = 4.57$
	$Ratio(C_{2,9}) = 5.55$	$Ratio(C_{3,14}) = 4.97$

At every iteration, the sub-chain with the minimum ratio is selected and its corresponding documents are added to $\prod$. Since sub-chain $C_{3,12} = d_{10}\ d_{11}\ d_{12}$ has the smallest sub-chain ratio, it is chosen initially. The documents of the chosen sub-chain are removed from the corresponding chain, and the chain ratio is recomputed. The steps are repeated until every document has been added to $\prod$. The document ordering generated for this example based on the interleaved chain ordering policy is d_{10}, d_{11}, d_{12}, d_5, d_6, d_1, d_2, d_3, d_4, d_7, d_8, d_9, d_{13}, d_{14}.

Theorem 3.3. *Algorithm 3.3 generates an optimal ordering for documents with precedence constraints using an interleaved document ordering policy in $O(n^2)$.*

Proof. To prove the optimality of Algorithm 3.3, it is enough to show that there exists an optimal permutation that contains the sub-chain with minimum ratio without interleaving any documents in the sub-chain. Without loss of generality, lets assume sub-chain $S = d_1\ d_2\ d_3\ d_4$ has the minimum ratio. Since S has minimum ratio, $\text{Ratio}(d_1\ d_2) \geq \text{Ratio}(d_1\ d_2\ d_3\ d_4)$, i.e.,

$$\frac{T_1+(1-P_1)T_2}{1-(1-P_1)(1-P_2)} \geq \frac{T_1+(1-P_1)T_2+(1-P_1)(1-P_2)T_3+(1-P_1)(1-P_2)(1-P_3)T_4}{1-(1-P_1)(1-P_2)(1-P_3)(1-P_4)}$$

Cross-multiply the ratios in the above expression and simplify. Thus, Ratio(d_1 d_2) $\geq$ Ratio(d_3 d_4).

Now, suppose there exists an optimal permutation that starts with d_1 d_2 **S'** d_3 d_4, where S' denotes another sub-chain of documents. By Lemma 3.2, Ratio(d_1 d_2) $\leq$ Ratio(S') $\leq$ Ratio (d_3 d_4). Previously, it was deduced that Ratio(d_1 d_2) $\geq$ Ratio(d_3 d_4), thus it is concluded that Ratio (d_1 d_2) = Ratio(S') = Ratio($d_3 d_4$). By the same lemma, sub-chain S' can be interchanged with $d_3 d_4$ and still obtain an optimal solution. Therefore, there exists an optimal permutation that contains the sub-chain d_1 d_2 d_3 d_4. □

3.4 Optimal Selection Problem

In section addresses the document selection problem which finds a subset of documents in the XML database that maximize the expected number of matches under a given upper bound on total processing time, T_{max}. Formally, find a subset of documents S = $\{d_{i1}, d_{i2}, \ldots, d_{ik}\}$

$$maximize \sum_{j=1}^{k} j \times prob(S \text{ has exactly } j \text{ matches}) \tag{3.6}$$

$$\text{such that } \sum_{j=1}^{k} T_{d_j} \leq T_{max}$$

Prob(S has exactly j matches) is expressed in terms of the document probabilities. For example, *prob(S has exactly 1 match)* = $P_{d_{i1}}$ (1 - $P_{d_{i2}}$) (1 - $P_{d_{i3}}$) ... (1 - $P_{d_{ik}}$) + (1 - $P_{d_{i1}}$)$P_{d_{i2}}$ (1 - $P_{d_{i3}}$) ... (1 - $P_{d_{ik}}$) + ... + (1 - $P_{d_{i1}}$)(1 - $P_{d_{i2}}$) (1 - $P_{d_{i3}}$) ... $P_{d_{ik}}$

The selection problem is a variation of the NP-complete Subset Sum Problem [25], since both problems try to find a subset S' where the sum of the elements in S' is exactly equal or does not exceed a target value *t*. The selection problem tries to find a subset S' whose total processing time does not exceed t, but also places an additional constraint which tries to maximize the number of matched documents. A naive solution computes all subsets whose total processing times does not exceed the target processing time t and then selects among the subsets one that maximizes the number of matches. Clearly, this algorithm has exponential time complexity. Thus, an effective heuristic algorithm is devised which invokes Algorithms 3.1 or 3.2 or 3.3. The heuristic algorithm devised for the selection problem is presented in Algorithm 3.4.

Algorithm 3.4: :Document Selection

Input:
T_{max} */* upper bound on total processing time */*
Output:
docSubset */* subset of documents */*

```
1 docSubset ← ∅;
2 time ← 0;
3 ∏ ← invoke Algorithm 3.1 or 3.2 or 3.3;
4 while time ≤ T_max do
5 |   add ∏(i) to docSubset;
6 |   time = time + T(∏(i) )  /* add processing time of i^th document
  |   */;
7 end
```

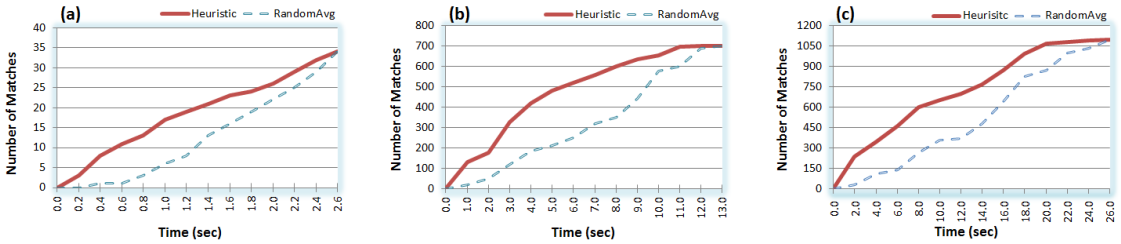


Figure 3.2: Performance results for document ordering and selection using for three XPath queries (a) Q_1, (b) Q_2, (c) Q_3.

3.5 Experimental Evaluation

Experimental Setup: The Swissprot dataset [1] was used to generate a large collection of XML documents of various sizes and features. The generated collection contained 300,000 documents which ranged between 50KB - 5MB in size. The documents were parsed and the statistics used to model the probability of having a query match and the document processing time were collected. The query evaluation was performed using LCSTRIM [77]. The LCSTRIM source code was updated to utilize an augmented index of XML terms, where terms were defined as values, tags, parent-child (or ancestor-descendant) pairs, and linear paths. A set of XPath queries were generated based on the Swissprot dataset. The experimental results are provided for the following three queries:

Q_1 - //Entry[Org][PFAM[@prim id=PF00304]][//SIGNAL/Descr],

Q_2 - //Entry[Org='Eukaryota'][Org='Metazoa'] [Org='Chordata'],

Q_3 - //Features[//'165'][/*/to][//'POLY-PRO'].

Experimental Results: Algorithms 3.1,3.2, and 3.3 were proven to provide an optimal ordering which minimizes the expected time to the first matched document. Experimental results are provided to empirically prove that these algorithms also minimize the expected

[1] http://www.cs.washington.edu/research/xmldatasets/

time to the first k matches, given that k is a number much less than the total number of matched documents. Furthermore, experimental results show that the algorithm proposed for the selection problem is an effective heuristic.

The performance is evaluated by measuring the number of matched documents obtained over time; the time includes the candidate document lookup time, the time for document ordering and selection, and the query processing time. Experimental results are provided for the ordering and selection problems for three different queries in Figures 3.2 a-c, using Algorithm 3.1 to obtain an ordering of candidate documents and Algorithm 3.4 to select a subset of candidate documents which satisfy the upper bound on processing time. Other experimental results for document collections with precedence constraints have shown to provide similar results and thus are omitted. The number of candidate documents for queries Q_1, Q_2, and Q_3 were 1,012, 1,852, and 2,643 documents, respectively. The y-axis in Figures 3.2 a-c denotes the number of matched documents obtained, while the x-axis denotes the processing time. A set of 1000 randomly selected subsets of documents were generated, and the average number of matches obtained by the generated subsets is plotted by 'randomAvg.' Note, the 'heuristic' plot, generated by Algorithms 3.1 and 3.4, performs better than any of the random subsets generated, but for simplicity only the average performance of the randomly generated subsets and permutations is plotted.

Consider Figure 3.2 (a), utilizing the document ordering generated by Algorithm 3.1, the first 10 matches were generated in 0.4 seconds. On the other hand, a random ordering of the documents required 1.2 seconds to generate 10 matches. Additional results consistently show that our document ordering algorithm minimizes the expected time to

the first k matches. As k approaches the maximum number of matches, the processing time of all permutations will be the same. For this reason, our algorithms are effective when k is relatively small compared to the total number of matches.

Now consider the number of matches generated within an upper bound of 1.2 seconds on processing time in Figure 3.2 (a). The 'heuristic' plot, which provides the results for Algorithms 3.1 and 3.4, generates 20 matches within that time bound, whereas, the 'randomAvg' plot generates only 10 matches. The area difference between 'heuristic' and 'randomAvg' denotes the performance gain by our selection and ordering algorithms. As shown by the results, our heuristic algorithms perform well in practice. The results also support the validity of the two estimation models for computing the probability of query match and the document processing time.

3.6 Final Remarks

This chapter addressed two optimization problems, ordering and selection of candidate documents for on-line query answering. For these problems, the objective functions were presented and expressed in terms of two parameters, the probability of a query having a match in a document and the document processing time. Experimental results further validate the proposed models for estimating the two parameters. Polynomial time algorithms were presented for the document ordering problem with and without precedence constraints. The algorithms were proven to provide optimal ordering which minimizes the expected time to the first matched document. Furthermore, experimental results showed that in practice the orderings generated by the algorithms also minimize

the expected time to the first k matched document, where k is a value much less than the total number of matches. The document selection problem was investigated which deals with finding a subset of documents that maximizes the expected number of matched documents for a given upper bound on total processing time. Since the document selection problem is NP-complete, a heuristic algorithm was devised. Empirical results have shown that this algorithm is effective heuristic.

Chapter 4

XML Structural Processing

4.1 Introduction

The eXtensible Markup Language (XML) has been ubiquitously adopted as the standard of data representation and exchange for a wide spectrum of applications, ranging from medical data to sensor data to news feeds. With this trend, research has targeted three important problems: XML filtering, XML query processing, and XML tuple- extraction.

In XML filtering, given a collection of queries/profiles (expressed in XPath), the objective is to identify those profiles that have a match in a given streaming XML document. When considering XML filtering, we are not interested in the location of the matches or in the number of matches, but only whether there exists a match in the XML document. Consider the XML tree and query twig shown in Figures 4.1a and 4.1b, respectively. A filtering system will simply return 'true' indicating the query has at least one match in T.

The aim of query processing is to find all matches in a XML document (or collection of XML documents) that satisfy a given query. Twig pattern matching returns a list of tuples, where each tuple consists of a node label (tag) and its unique node-id (preorder). Generally, structural summaries or indexes are pre-built on the XML database to speed-up query processing by locating relevant XML nodes quickly. Using the example in Figure 4.1, the query will return two matches in the XML tree, r_1 and r_2.

The XML tuple-extraction problem differs from traditional XML querying in that no indexes exist over the XML data and the stream is processed only once. Extraction queries are specified as twig queries with a single or multiple extraction node(s). If a node is specified as an 'extraction node', the value (or leaves) of the node should be returned if the twig pattern is satisfied. In Figure 4.1 the node 'booktitle' is specified as an extraction node with the symbol #. The result set is simply two tuples, rs_1 and rs_2, containing the value of the extraction node.

The algorithms presented concentrate on *ordered* query (or twig pattern) matching, i.e. twig pattern nodes must follow the XML document order. Several works have addressed the importance and applications of ordered twig pattern matching [41, 64, 63]. For example, [64, 63] discuss various linguistics applications which require ordered twig matching. Another application is biological databases, which are largely adopting XML for data representation and require ordered-based twig matching. For example, to encode a particular protein, the order of the amino acids is very important and a result that does not match the order specified in the query may be irrelevant to the user. Order is also important in multimedia applications like querying music objects [3]. Previous

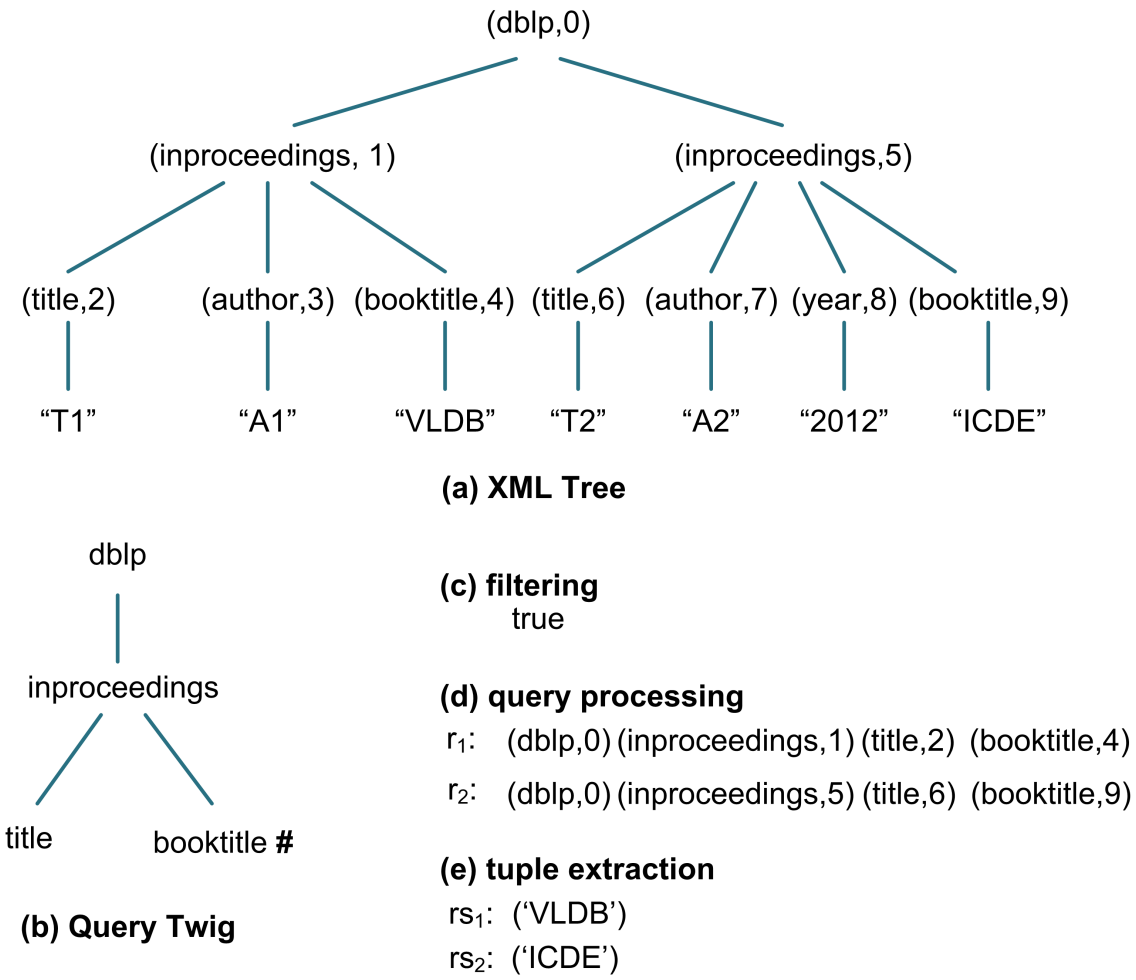


Figure 4.1: XML Tree and XPath Query Example

work on ordered twig matching [82, 64, 63, 41, 42, 78, 50, 73, 86] transform the XML tree and queries into sequences, then employ subsequence matching and verification to determine true matches. These sequence-based approaches have an advantage over those approaches that support unordered query matching, since they inherently support ordered twig matching and hence no extra work is performed during or after the matching phase to remove those matches that do not follow the query order.

Here a sequence-based approach is faciliated which represents the query and

XML document using a sequential encoding referred to as Node Encoded Tree Sequence (NETS). The query matching is composed of two processes: subsequence matching and structural matching, which can be performed concurrently or sequentially depending on the problem. The solution of the subsequence matching problem is based on the classical dynamic programming recurrence relation. For structural matching, a new necessary and sufficient condition is presented which provides a simple verification procedure.

In this chapter, a unified approach will be presented to support ordered twig matching for the three XML processing problems. The key contributions are summarized as follows:

- A new necessary and sufficient condition is presented for an ordered query to have a match in a given XML tree using the Node Encoded Tree Sequence (NETS) for representing XML documents and queries.

- The dynamic programming recurrence relation for the Longest Common Subsequence (LCS) problem is combined with the new necessary and sufficient condition to formulate a new recurrence relation which can be used to find all query matches in the XML document.

- The fundamental concepts and procedures of the unified twig matching approach is presneted of our unified approach used to devise efficient algorithms for all three problems. The algorithms are composed of two verification procedures, subsequence matching and structural matching, which can be executed concurrently or sequentially.

- An efficient algorithm for the XML filtering and tuple-extraction problems is presented, where subsequence and structural matching are performed concurrently (Forward-Match

Algorithm).

- An efficient algorithm for the qury processing problem is presented, which performs subsequence and structural matching sequentially and utilizes a compact graph representation of all potential subsequence matches (Backward-Match Algorithm).

- The algorithms were shown to outperform state of-the-art approaches for each of the three XML processing problems.

Section 4.2 will outline related work. The fundamental concepts and theory of our unified approach are presented in Section 4.3. Section 4.4 provides the details of the stream matching algorithm (Forward-Match) for the XML filtering and tuple-extraction problems. Section 4.5 provides the details of a new algorithm for query processing referred to as Backward-Match. Section 4.6 presents experimental results while final remarks appear in Section 4.7.

4.2 Related Work

Much work has been presented on the three XML matching problems. Previous approaches can be classified based on the three problems.

Previous approaches presented for the query processing problem can be broadly classified into two categories: *path-based approaches* and *holistic approaches*. Path-based approaches generally decompose a twig pattern into root-to-leaf paths and then apply a matching algorithm to match the individual paths. The individual solutions are then joined to produce the final answer set [7, 22, 46]. Typically, holistic approaches outperform

path-based ones as they avoid the expensive join operation by treating the whole twig as the base unit of matching. Many works within this category are based on TwigStack [13].

Another main approach of holistic algorithms transforms the problem to subsequence matching by encoding the trees as sequences thus inherently supporting ordered twig matching [64, 63, 78, 83, 86]. Wang et al proposed ViST [83] that transforms the XML tree into sequences based on pre-order traversal. Rao et al proposed PRIX [64, 63] which transforms the XML into Prufer sequences which are constructed from based on post-order traversal of the tree. LCS-TRIM [78] proposed a consolidated Prufer sequence encoding and applied dynamic programming to compute the LCS matrix for the XML and query sequences. LCS-TRIM has been shown to outperform other approaches for ordered twig query matching and thus it shall be used as a competitor for experimental evaluation. Our approach differs from these methods in several ways. First, the NETS encoding is easy to generate and update. Second, the necessary and sufficient condition required for a true match is simple to verify compared to the other sequence encodings. Third, the algorithms proposed for NETS matching identifies false positives early in the matching phase and does not require a post-processing phase.

Numerous works have been presented for the node-extraction problem [39, 59, 60, 61, 32, 36]. In [32], two stack-based stream querying algorithms are presented, namely, LQ (lazy querying) and EQ (eager querying) for single-node extraction queries. More recently, StreamTX [36] presented an approach that extends TwigStack for the multiple-node extraction problem which is able to extract nodes from a streaming document. While these algorithms can be extended to support ordered twig matching, their primary focus is on unordered twig matching.

Table 4.1: List of notations

T	database tree with n nodes
Q	twig pattern with m nodes
$NETS_T$	Node Encoded Tree Sequence of T (similarly for $NETS_Q$)
$NETS_T$[k]	k^{th} symbol in $NETS_T$
$NETS_T$ [1...k]	substring of $NETS_T$ from index 1 to k
preorder($NETS_T$[k])	preorder number of node associated with $NETS_T$[k]
level($NETS_T$[k])	level of node associated with $NETS_T$[k]
R[i][j]	LCS length of $NETS_Q$[1...i] and $NETS_T$[1...j]
B[i][j]	list of bitsets of subsequence matches of $NETS_Q$[1...i] and $NETS_T$[1...j]
Z	bitset where all bits are zero

XML filtering is a core component of Publish-Subscribe systems and many papers have been presented on this problem [26, 41, 73, 43, 15, 52, 5, 62]. The approaches proposed for XML filtering primarily fall into two categories: FSM-based, sequence-based. YFilter [26] builds a single NFA that combines all user profiles into a single machine, thus exploiting the commonality among path expressions. The twig profiles are broken into simple linear paths, thus requiring an expensive post-processing phase to join the results. FiST [41, 43] proposed a filtering system which encodes both the XML document and query profiles into Prufer sequences and performs subsequence matching, but also required a post-processing phase to filter false positives.

4.3 Unified Approach

In this section, the definition of NETS is presented and a necessary and sufficient condition for structural matching is introduced.

4.3.1 Node Encoded Tree Sequence (NETS) Representation

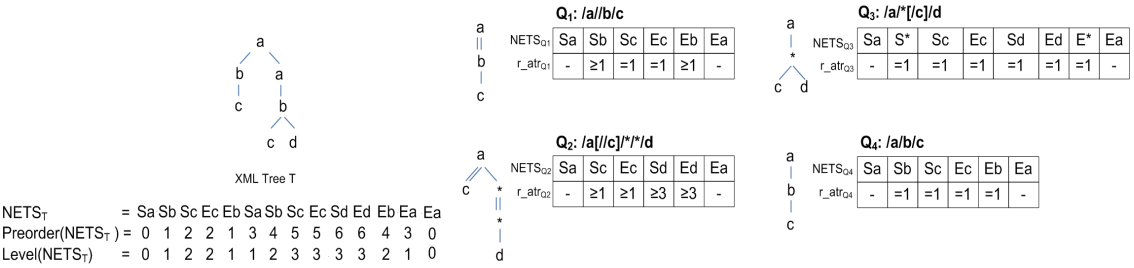


Figure 4.2: XML tree T and twig queries Q_1, Q_2, Q_3, and Q_4 and their NETS representation.

An XML document is modeled as a rooted ordered labeled tree where each node corresponds to an element tag, attribute, or value, and edges represent structural relationships between nodes. Values are represented by character data and occur at the leaf nodes. For each node n ∈ T, the Node Encoded Tree Sequence, $NETS_T$, contains two symbols referred to as 'start-symbol' and 'end-symbol'. The start-symbol and end-symbol of the same node are called *corresponding symbols*. If a node is labeled by x, the preorder and level of the corresponding symbols Sx and Ex are defined as preorder(Sx)=preorder(Ex)=preorder(x) and level(Sx)=level(Ex)=level(x), respectively. The NETS of tree T is formally defined in Definition 4.1 and an example is illustrated in Figure 4.2.

Definition 4.1. ***(Node Encoded Tree Sequence):***

- *If tree T consists of a single node labeled by r, then the $NETS_T$ is Sr Er.*
- *Let r be the root of tree T and assume r has n children labeled from left-to-right as r_1, r_2, ..., r_n. Let sequences S_1, S_2, ..., S_n be the NETS of subtrees whose roots are r_1,*

r_2, ..., r_n, *respectively. Then the* $NETS_T$ *is* $Sr\ S_1\ S_2\ \ldots S_n\ Er$.

A twig pattern is expressed using a core fragment of the XPath language. XPath query language considers the inherent tree structure thus enabling querying on the document structure as well as simple values. While value-based conditions can be performed efficiently using traditional indexing structures, evaluating structural conditions of the query is more challenging. The grammar of the supported query language is given in Table 4.2. The query consists of a sequence of location steps, which includes wildcard node test ('*'), joined by '/' or '//', where '/' denotes the child location step and '//' denotes the descendant location step. The NETS of a query twig Q, NETS_Q, is generated in accordance to Definition 4.1. For each start-symbol and end-symbol in NETS_Q, an additional attribute referred to as 'relationship-attribute', r_atr_Q, is encoded to specify the relationship between a node and its parent. For example, if the relationship is parent-child ('/') then the attribute shall be '=1', specifying that two nodes must be *exactly* one level apart. Whereas, if the relationship is ancestor-descendant ('//') then the attribute shall be '$\geq$', specifying that two nodes must be at *least* one level apart. Wildcards are encoded differently depending on the occurrence of the '*' in the query. If the wildcard appears as a branch node in the twig, then the '*' is encoded as a regular node (see Q_3 in Figure 4.2). Otherwise, if it is a non-branch node, then the next non-wildcard node is encoded in NETS_Q and the occurrence of the wildcard is reflected by r_atr_Q (see Q_2 in Figure 4.2). To determine whether a given twig pattern Q has a 'match' in a XML tree T, the twig pattern Q must be a subgraph of the XML tree, which is formally define below:

Definition 4.2. ***(Subgraph):*** *Let T=(V,E) be a labeled tree, where V is the set of all*

Table 4.2: Subset XPath query grammar

Query	Step | Query Step
Step	Axis NodeTest | Axis NodeTest '['Predicate']' +
Axis	'/' | '//'
NodeTest	String | '*'
Predicate	Query | Query '=' String

nodes in T and E is the set of all edges in T. For every node n ∈ V, let label(n) *denote the label of n and* preorder(n) *denote the unique id based on preorder traversal of the tree. A labeled tree Q=(V',E') is a subgraph of T if the following three conditions hold:*

1. *There is a one-to-one mapping f() from V' into V such that for every node n ∈ V' label(n)=label(f(n))*

2. *For every edge (n_1,n_2) in E' there is a path from f(n_1) to f(n_2) in T, such that:*

- *if edge is '/', then level(f(n_1))−level(f(n_2)) = 1.*
- *if edge is '//', then level(f(n_1))−level(f(n_2)) ≥ 1*

3. *For every two nodes n_1 and n_2 in Q, if preorder(n_1) < preorder(n_2) then preorder(f(n_1)) < preorder(f(n_2)). This condition guarantees that the relative order of nodes in Q corresponds to the one in T.*

Our query matching approach operates on the NETS representation of a database tree T and a query Q. The following theorem states the relation between the query twig and XML tree and their sequence representation.

Theorem 4.1. *Given two rooted labeled trees T and Q, if Q has a match in T (i.e., satisfy Subgraph definition) then $NETS_Q$ is a subsequence of $NETS_T$ [73].*

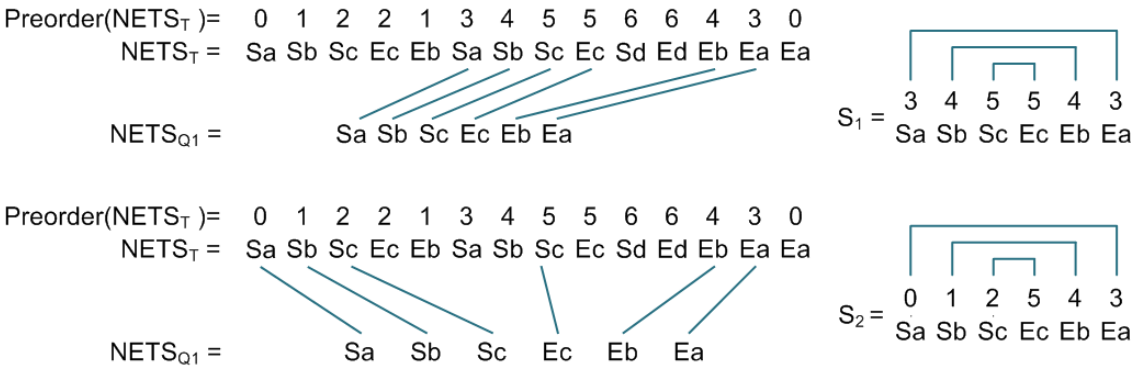


Figure 4.3: Example illustrates false match and preorder consistent property

The above theorem provides a necessary condition for a query to have a match in T in terms of NETS representation; i.e., if all possible subsequences of NETS$_T$ that match NETS$_Q$ are enumerated, then all matches are guaranteed to be reported with no false dismissals. However, note that the result may contain false positives, since the condition of Theorem 4.1 is not a sufficient condition. For example, consider XML tree T and query twig Q_1 shown in Figure 4.2. Figure 4.3 shows two subsequences of NETS$_T$, namely, S_1 and S_2. The first sequence, S_1, represents a true match of Q_1, whereas the second sequence, S_2, is not a true match since the preorder numbers of the start-symbols and end-symbols are not equal. Thus, a necessary and sufficient condition for a true match will be presented.

4.3.2 Necessary and Sufficient Condition for Query Matching

First, two properties are defined: Level-Consistent Subsequence (Definition 4.3) and Preorder-Consistent Subsequence (Definition 4.4), which are part of the sufficient condition required for a true match.

Definition 4.3. ***(Level-Consistent Subsequence):*** *A subsequence S of $NETS_T$ is level consistent with respect to Q if:*

1. *$S = NETS_Q[1 \ldots i]$ where $i \leq |NETS_Q|$.*
2. *For each edge (n_1,n_2) in Q such that the start-symbols of n_1 and n_2 are in $NETS_Q[1 \ldots i]$, the following property holds:*
   - *if edge is of type '//' then level(Sy)-level(Sx) $\geq$ 1.*
   - *if edge is of type '/', then level(Sy)-level(Sx) = 1,*

   where Sx and Sy are the start-symbols in S and correspond to start-symbols of n_1 and n_2 in $NETS_Q$.

Definition 4.4. ***(Preorder-Consistent Subsequence):*** *A subsequence S of $NETS_T$ is preorder consistent if for each end-symbol 'Ex' in S, the* corresponding *start-symbol 'Sx' is also in S. Start-symbol 'Sx' and end-symbol 'Ex' are said to be* corresponding symbols *if preorder(Ex) = preorder(Sx).*

Example 4.1. Consider XML tree T and query twig Q_4 shown in Figure 4.2. Figure 4.4 shows two subsequences of $NETS_T$, namely, S_1 and S_2. The first sequence, S_1, represents a level-consistent subsequence since it satisfies the r_atr of $NETS_{Q4}$. Whereas the second sequence, S_2, is not a level-consistent subsequence since the r_atr is not satisfied for start-symbol 'Sa'. Now consider XML tree T and query twig Q_1 shown in Figure 4.2. Figure 4.3 shows two subsequences of $NETS_T$, namely, S_1 and S_2. The first sequence, S_1, represents a preorder consistent subsequence, whereas the second sequence, S_2, does not satisfy this property since the preorder of start-symbol 'Sc' does not equal that of end-symbol 'Ec'.

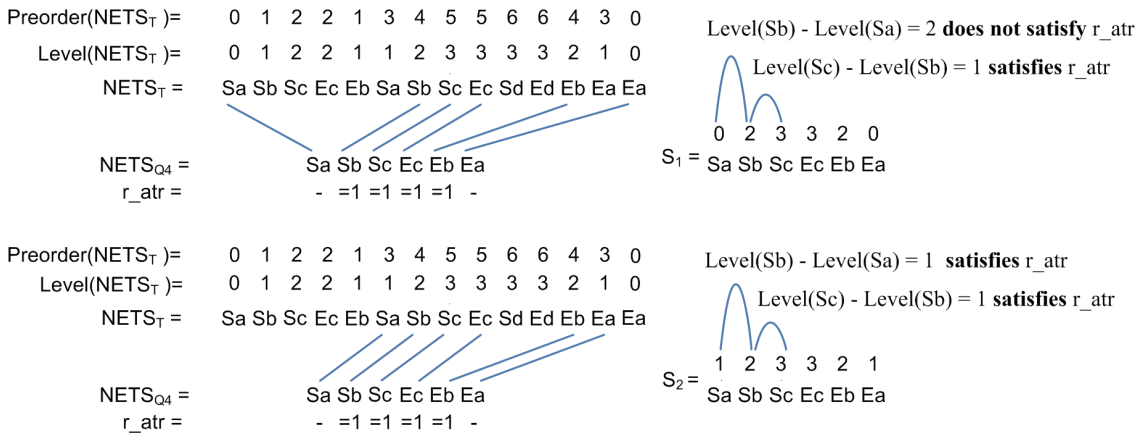


Figure 4.4: Example illustrates level-consistent subsequence property

Theorem 4.2. *Given two rooted labeled trees Q and T, Q has a match in T iff there is a subsequence S of NETS$_T$ such that:*

1. *S is equal to NETS$_Q$.*
2. *S is a level-consistent subsequence with respect to Q (Definition 4.3).*
3. *S is a preorder-consistent subsequence (Definition 4.4).*

Proof. ***Necessary Condition:*** Assume a query Q has a match in the tree T. This implies Q is a subgraph of T and NETS$_Q$ is a subsequence of NETS$_T$ (Theorem 4.1). Let S be a subsequence of NETS$_T$ and let S equal NETS$_Q$. Thus, the first condition of Theorem 4.2 is satisfied. If end-symbol 'Ex' is in S, then its corresponding start-symbol 'Sx' must also be in S and consequently the preorder-consistent property (condition 3) is satisfied. This follows from the fact that the symbols in NETS$_T$ which are in S are determined by the one-to-one mapping function given in (Definition 4.2) from the nodes of Q into the nodes of T. The third condition (level-consistent property) follows from the second property of Definition 4.2. Thus, S satisfies the three conditions of Theorem 4.2.

Sufficient Condition: Assume that there exists a subsequence S of NETS_T which satisfies the three conditions of Theorem 4.2. To prove that Q is a subgraph of T; i.e., prove there is a one-to-one mapping f() from the nodes of Q into the nodes of T which satisfies the definition of Subgraph (Definition 4.2). Let n be a node of Q with label x. Since S is a preorder-consistent subsequence of NETS_T and S=NETS_Q, the start-symbol Sx and end-symbol Ex of n must appear in S. Next, it will be shown that these two symbols in S correspond to the start-symbol and end-symbol of the same node in T. This node will be used to define the one-to-one mapping function f(n). Two cases must be considered:

Case 1: *There is only one node in Q with label x.*

Since S is a preorder-consistent subsequence, S contains only one Sx and one Ex and they must be the start-symbol and end-symbol of the same node in T (referred to as p). Thus, f(n) is defined to be p.

Case 2: *Q has more than one node with label x.*

Without loss of generality, assume Q has only two nodes (n and m) with label x, where preorder(n) < preorder(m). If there is a path from n to m in Q, then the start-symbols and end-symbols of n and m must appear in the following order:

$$\begin{array}{cccc} n & m & m & n \\ Sx\ldots & Sx\ldots & Ex\ldots & Sx \end{array} \quad (nested\ case)$$

If there is no path from n to m, then the start-symbols and end-symbols of n and m must appear in the following order:

$$\begin{array}{cccc} n & n & m & m \\ Sx\ldots & Ex\ldots & Sx\ldots & Ex \end{array} \quad (disjoint\ case)$$

Since S = NETS_Q and S is a preorder-consistent subsequence, then S contains two start-symbols Sx and two end-symbols Ex which correspond to two nodes (p and

q) of T. For the nested cases, there are four possible orderings of the start-symbols and end-symbols of nodes p and q.

	p	q	q	p		q	p	p	q
1)	$Sx...$	$Sx...$	$Ex...$	Ex	2)	$Sx...$	$Sx...$	$Ex...$	Ex
	p	q	p	q		q	p	q	p
3)	$Sx...$	$Sx...$	$Ex...$	Ex	4)	$Sx...$	$Sx...$	$Ex...$	Ex

The last two orderings are not possible because the start-symbol and end-symbol of p and q must be either disjoint or nested. The first two orderings show that the start-symbol and end-symbol of n are associated with a single node in T, either p or q. This mapping defines f(n)=p or f(n)=q. Similarly, four possible orderings for the disjoint case are considered.

	p	p	q	q		q	q	p	p
1)	$Sx...$	$Ex...$	$Sx...$	Ex	2)	$Sx...$	$Ex...$	$Sx...$	Ex
	p	q	q	p		q	p	p	q
3)	$Sx...$	$Ex...$	$Sx...$	Ex	4)	$Sx...$	$Ex...$	$Sx...$	Ex

The last two orderings are excluded because the end-symbol of a node (p or q) appears before its start-symbol. Consequently, the Sx and Ex of n are associated with a single node in T. For both cases, if f(n)=p then f(m)= q or if f(n)=q then f(m)= p, thus, according to the possible orderings the preorder(f(n)) < preorder(f(m)) and consequently the third property of Definition 4.2 is satisfied. The second condition of Theorem 4.2 corresponds to the second condition of Definition 4.2, thus it is simple to verify that Q satisfies the level-consistent property. Since all conditions of the theorem are satisfied, Q must have a match in T. □

As described previously, the query matching algorithms are composed of two procedures. The first condition of Theorem 4.2 relates to the subsequence matching

procedure, while the second and third conditions of the theorem relate to the structural matching procedure. To determine if NETS_Q is a subsequence of NETS_T, the classical dynamic programming algorithm [81] is applied to compute the length of the Longest Common Subsequence (LCS) of $\text{NETS}_Q[1 \ldots \text{m}]$ and $\text{NETS}_T[1 \ldots \text{n}]$, where m and n are the length of NETS_Q and NETS_T, respectively. The algorithm computes matrix R per Recurrence Relation 4.1, where entry R[i][j] represents the LCS length of $\text{NETS}_Q[1 \ldots \text{i}]$ and $\text{NETS}_T[1 \ldots \text{j}]$. Consequently entry R[m][n] gives the LCS length of NETS_Q and NETS_T. If entry R[m][n] is equal to the length of NETS_Q, then NETS_Q is said to be a subsequence of NETS_T.

Recurrence-Relation 4.1. *R[i][j] =*

$$\begin{cases} \textbf{\textit{IF}}\ i = 0\ \textbf{\textit{OR}}\ j = 0 & ;\ 0 \\ \textbf{\textit{IF}}\ NETS_Q[i]\ =\ NETS_T[j] & ;\ R[i-1][j-1] + 1 \\ \textbf{\textit{IF}}\ NETS_Q[i]\ \neq\ NETS_T[j] & ;\ max(R[i][j-1], R[i-1][j]) \end{cases}$$

Next, the structural matching verification procedure is discussed. A bitset structure is associated with each partial subsequence match. It provides a simple mechanism for verifying the level-consistent and preorder-consistent properties of Theorem 4.2 required for a true match. Furthermore, the bitset structure of a subsequence provides a condensed representation of the current state of match. The definition of a bitset structure is given below:

Definition 4.5. ***(Bitset b_S):*** *Let S be a subsequence of $NETS_T$ such that for every Ex in S the corresponding start-symbol Sx is also in S. The bitset of S is denoted by b_S. Note,*

		1	2	3	4	5	6	7	8	9	10	11	12
		Sa Level = 0 Preorder = 0	**Sb** Level = 1 Preorder = 1	**Sc** Level = 2 Preorder = 2	**Ec** Level = 2 Preorder = 2	**Eb** Level = 1 Preorder = 1	**Sa** Level = 1 Preorder = 3	**Sb** Level = 2 Preorder = 4	**Sc** Level = 3 Preorder = 5	**Ec** Level = 3 Preorder = 5	**Eb** Level = 2 Preorder = 4	**Ea** Level =1 Preorder = 3	**Ea** Level = 0 Preorder = 0
1	**Sa**	R=1 B={(0)}	R=1 B={(0)}	R=1 B={(0)}	R=1 B={(0)}	R=1 B={(0)}	R=1 B={(0), (3)}	R=1 B={(0), (3)}	R=1 B={(0), (3)}	R=1 B={(0), (3)}	R=1 B={(0), (3)}	R=1 B={(0), (3)}	R=1 B={(0), (3)}
2	**Sb**	R=1 B=Ø	R=2 B={(0,1)}	R=2 B={(0,1)}	R=2 B={(0,1)}	R=2 B={(0,1)}	R=2 B={(0,1)}	R=2 B={(0,1) (0,4), (3,4)}	R=2 B={(0,1) (0,4), (3,4)}	R=2 B={(0,1) (0,4), (3,4)}	R=2 B={(0,1) (0,4),(3,4)}	R=2 B={(0,1) (0,4), (3,4)}	R=2 B={(0,1) (0,4), (3,4)}
3	**Sc**	R=1 B=Ø	R=2 B=Ø	R=3 B={(0,1,2)}	R=3 B={(0,1,2)}	R=3 B={(0,1,2)}	R=3 B={(0,1,2)}	R=3 B={(0,1,2)}	R=3 B={(0,1,2), (0,4,5), (3,4,5)}	R=3 B={(0,1,2), (0,4,5), (3,4,5)}	R=3 B={(0,1,2), (0,4,5), (3,4,5)}	R=3 B={(0,1,2), (0,4,5), (3,4,5)}	R=3 B={(0,1,2), (0,4,5), (3,4,5)}
4	**Ec**	R=1 B=Ø	R=2 B=Ø	R=3 B=Ø	R=4 B={(0,1)}	R=4 B={(0,1)}	R=4 B={(0,1)}	R=4 B={(0,1)}	R=4 B={(0,1)}	R=4 B={(0,1) , (0,4), (3,4) }	R=4 B={(0,1) , (0,4), (3,4) }	R=4 B={(0,1) , (0,4), (3,4) }	R=4 B={(0,1) , (0,4), (3,4) }
5	**Eb**	R=1 B=Ø	R=2 B=Ø	R=3 B=Ø	R=4 B=Ø	R=5 B={(0)}	R=5 B={(0)}	R=5 B={(0)}	R=5 B={(0)}	R=5 B={(0)}	R=5 B={(0), (3)}	R=5 B={(0), (3)}	R=5 B={(0), (3)}
6	**Ea**	R=1 B=Ø	R=2 B=Ø	R=3 B=Ø	R=4 B=Ø	R=5 B=Ø	R=5 B=Ø	R=5 B=Ø	R=5 B=Ø	R=5 B=Ø	R=5 B=Ø	R=6 B={(Z)}	R=6 B={(Z), (Z)}

Figure 4.5: The R and B matrix for trees T and Q_1 of Figure 4.2 is computed according to Recurrence Relation 4.1 (for matrix R) and Recurrence Relation 4.2 (for matrix B).

a bitset b whose bits are all set to zero is represented as Z. The state of a bit at index i, given by $b_S(i)$, is defined below:

$$b_S(i) = \begin{cases} \textbf{IF } Sx \in S \textbf{ AND } Ex \in S, where & ; \ b_S = 0 \\ preorder(Sx) = preorder(Ex) = i & \\ \textbf{IF } Sx \in S \textbf{ AND } Ex \notin S, where & ; \ b_S = 1 \\ preorder(Sx) = preorder(Ex) = i & \end{cases}$$

Example 4.2. *Consider two subsequences S_1 and S_2 of $NETS_T$ that match $NETS_{Q_1}$ in Figure 4.2, the corresponding bitsets b_{S1} and b_{S2} of the two subsequences are given as follows per Definition 4.5.*

The algorithms presented utilize a bitset structure for the structural matching verification procedure, alternatively, a set of preorder numbers can be used as shown in the example. The set of preorder numbers is bounded in length by the query length. A set '1' bitset operation translates into an insert event of the start-symbol's preorder number at the end of the set. While, a set to '0' bitset operation translates to a delete event from the end of the set. The implementation of the algorithms are based on the set of preorder numbers.

The algorithms presented are based on computing matrix B per Recurrence Relation 4.2, where entry B[i][j] represents the list of bitsets of subsequence matches between $\text{NETS}_Q[1 \ldots \text{i}]$ and $\text{NETS}_T[1 \ldots \text{j}]$. If the entry B[m][n] contains a bitset b which is equal to Z (i.e., all bits in b are set to zero), then the sequence satisfies the sufficient conditions of Theorem 4.2 for a true match.

Recurrence-Relation 4.2. ***B[i][j] =***

$$
\begin{cases}
\textbf{\textit{IF}}\ i = 0\ \textbf{\textit{OR}}\ j = 0\ \textbf{\textit{OR}}\ R[i][j] < i & \textit{; } \oslash \\
\\
\textbf{\textit{IF}}\ R[i][j] = i\ AND & \\
\quad NETS_Q[i]\ =\ NETS_T[j]\ \textbf{\textit{AND}} & \textit{; copy \& update each} \\
\quad NETS_Q[1 \ldots i]\ \textit{is preorder} & \textit{bitsets b in } B[i-1][j-1] \\
\quad \&\ \textit{level} - \textit{consistent subsequence of}\ NETS_T[1 \ldots j] & \textit{and add to } B[i][j] \\
\textbf{\textit{IF}}\ R[i][j-1]\ =\ i\ \textbf{\textit{OR}} & \textit{; copy bitsets in} \\
NETS_Q[i]\ \neq\ NETS_T[j] & B[i][j-1]\ \textit{to}\ B[i][j]
\end{cases}
$$

The Recurrence Relation 4.2 computes entries B[i][i] of matrix B which is utilized for structural matching. The first case of the recurrence relation initially sets entry B[i][j] to be an empty list. Entry B[i][j] is also set to an empty list if the computed LCS length at R[i][j] is less than 'i' (i.e. $|NETS_Q[1 \ldots i]|$, because a subsequence of whose length is less than 'i' will not lead to whole match of subsequence NETS_Q.

The second case is executed if the sequence labels of NETS_Q[i] and NETS_T are equal, and NETS_Q[1 ...i] is a level-consistent and preorder-consistent subsequence of NETS_T[1...j]. The bitsets in diagonal entry B[i-1][j-1] are added to B[i][j] and each bitset b is updated as follows: If the label of NETS_T[j] is a start-symbol, the bit corresponding to the preorder(NETS_T[j]) is set to '1' to indicate that the start-symbol has been matched. Otherwise, if the label of NETS_T[j] is a end-symbol, the bit corresponding to the preorder(NETS_T[j]) is verified to be '1' then set to '0'. This condition verifies that the corresponding start-symbol has been matched then resets the bit to '0' to indicate its

corresponding end-symbol has been matched. If the bit is not set to '1', then structural matching fails for this subsequence and its bitset will be discarded.

The last case is executed if the sequence labels of NETS_Q[i] and NETS_T[j] are not equal or if entry R[i][j-1] equals i. This case is required to add the partial matches given by B[i][j-1] to B[i][j] in order to maintain the state of previous matches. Observe that the list of bitset in entry B[i-1][j] are not considered. This entry provides matches for $\text{NETS}_Q[1 \ldots \text{i-1}]$ whose subsequence length is less than i, and since the focus of query matching is to match $\text{NETS}_Q[1 \ldots \text{i}]$ in its entirety, entry B[i-1][j] can be ignored.

Example 4.3. Consider the XML tree T and query Q_1 shown in Figure 4.2. Matrix R and B for NETS_T and NETS_Q are illustrated as one matrix in Figure 4.3.2. The notation R&B[i][j] is used to denote the cells R[i][j] and B[i][j]. Observe that the entire matrix is shown for clarity, but only a **single column** should be maintained. For each NETS_Q[i] and NETS_T[j], the cell contains the value of R[i][j] denoted by R and the list of bitset B[i][j] denoted by B. Consider entry R&B[2][3] in the R and B matrix. The LCS length of $\text{NETS}_Q[1 \ldots 2]$ and $\text{NETS}_T[1 \ldots 3]$ is 2, which is given by R. Entry B[2][3] contains a single bitset (0,1), where the bits at indices 0 and 1 are set to '1', and all other bits are set to '0'. This denotes that two start-symbols, with preorder 0 and 1, have been matched but their corresponding end-symbols have not been matched. The first case of Recurrence Relation 4.2 is illustrated by entry R&B[3][2]. The LCS length given by R[3][2] is less than 3, thus, structural matching is skipped and entry B[3][2] is set to empty list. The second case of the relation is illustrated by entry R&B[2][7] where the symbols of NETS_Q[2] equal that of NETS_T[7], and the level-consistent and preorder-consistent properties hold. Here, the

bitsets in entry B[1][6] {(0),(1)} are updated to {(0,4),(1,4)} then added to entry B[2][7]. Furthermore, since the LCS length given by R[2][6] is equal to current entry R[2][7], the bitset in entry B[2][6] {(0,1)} is also added to B[2][7] per the third case of the recurrence relation. The third case is executed if the first two cases were not satisfied. This case is illustrated by entry R&B[2][3]. Here, $\text{NETS}_Q[2]$ does not equal $\text{NETS}_T[3]$, thus the bitset in entry B[2][2] {(0)} is added to B[2][3] in order to capture information about previous matches.

4.4 Filtering and Tuple-Extraction

In the first part of this section a simple XML filtering algorithm, referred to as NETS-Filter, is presented. In the second part of this section presents an efficient and simple to implement algorithm for the two streaming problems, namely filtering and tuple-extraction, called Forward Match.

NETS-Filter algorithm utilizes several structures for subsequence matching. A runtime global stack is maintained by the filtering algorithm where each entry in the stack is a tuple that contains a start-symbol of the XML sequence and its preorder and level. The tuples are pushed to the stack as start-symbols are generated. At the start of the filtering algorithm, concurrent subsequence searches are initiated over the query sequences. For non-recursive XML documents, only one subsequence search is needed for each query, however, for recursive XML, more than one subsequence search over the same query sequence may be initiated. Each subsequence search maintains queryPos and queryStack. The queryPos is an integer that denotes the current position in the

query sequence. The queryStack is a stack where each entry is an ordered pair. The first component in the stack is the index of the matched XML start-symbol in the global stack. The second component is the position of the matched startsymbol in the query sequence. The ordered pairs are pushed and popped to and from the queryStack as start and end symbols in the XML sequence are matched to the query sequence. The filtering algorithm also utilizes a dynamic hashtable, called sequenceIndex, to facilitate concurrent processing of query twigs. The sequenceIndex uses the start and end symbol assigned to each XML tag as a key into the hashtable. For each key (start or end symbol) it maintains a list of queries to be matched specified by their queryIds, the querys unique identification

At the start of filtering, the first node of each query sequence is inserted into the sequenceIndex. The streaming XML document is parsed by the SAX parser; the ProcessStartSymbol(.) function is called when an open tag is generated and the ProcessEndSymbol is called when an end tag is generated. Note that the SAX methods have been slightly altered to maintain level and preorder information for each XML tag.

The ProcessStartSymbol(.) function is described by Algorithm 4.1. This function receives a start-symbol and its preorder number and level as input. The filtering algorithm probes the sequenceIndex for a list of queries that match the startSymbol. For each query in the currentList, the algorithm verifies that the level-consistent property holds for the current query node. The top entry of the queryStack is retrieved and the first component, referred to as indexParAnc, is used to retrieve the xmlParAncNode from the globalStack. The difference between the current startSymbols level and the xmlParAncNodes level is calculated to determine whether the querys relationship attribute is satisfied. If the property is satisfied, then the following steps are performed. First, the

index of the matched startSymbol (its index in the globalStack) and the queryPos are inserted into the queryStack. Second, the queryPos is incremented by one. Lastly, the next symbol in the query sequence is added to the sequenceIndex.

The ProcessEndSymbol(.) function is described by Algorithm 4.1. For each query in the currentList, the filtering algorithm first verifies that the current endSymbols preorder number equals that of the xmlParAncNode. If there is a match, then the queryId is deleted from the start-symbols and end-symbols lists in the sequenceIndex. If the end of the query sequence is reached, then the query is reported as a match. Otherwise, the next symbol is added to the sequenceIndex. At times, backtracking to a previous query sequence position is required due to a false match (lines 16-23). The sequenceIndex is probed for the start-symbol and a list of queryIds is retrieved. For each query in currentList, the top entry of the queryStack is retrieved and the first component, referred to as indexParAnc, is used to retrieve the xmlParAncNode from the globalStack. The xmlParAncNodes preorder is compared to the current endSymbols preorder. If there is a preorder match, then backtracked is performed by the following steps. First, the top entry is popped off the queryStack. Second, the last inserted queryId is deleted from the sequenceIndex, and lastly the queryPos is updated to indicate the new position in the query sequence.

Below, we illustrate the execution of NETS-Filter algorithm with the XML tree T_2 and twig patterns Q_5 and Q_6 shown in Figure 4.6.

Example 4.4. *In Figure 4.4(a), the ProcessStartSymbol(.) is invoked for Sa and the sequenceIndex contains Q_5 and Q_6 for key Sa. The level-consistent property is automati-*

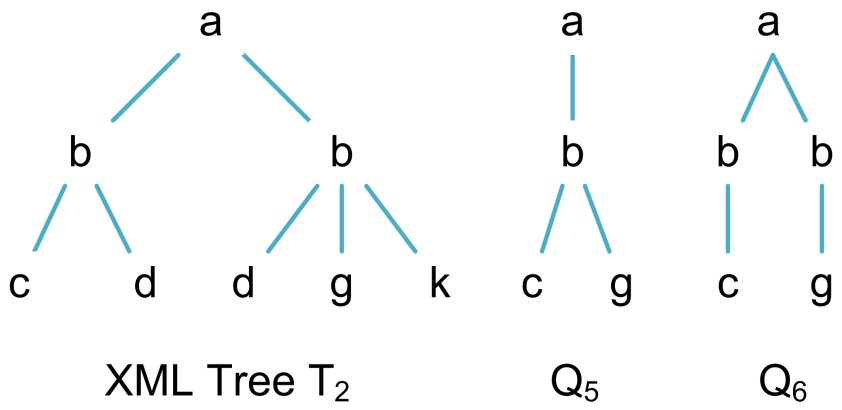


Figure 4.6: XML Tree T_2 and twig patterns Q_5 and Q_6

cally satisfied since the queryPos points to the first symbol in the query sequence. Thus, the subsequence search for both Q_5 and Q_6 is advanced. First, the next symbol in the query sequence (Sb for both Q_5 and Q_6) is retrieved and the queryIds are inserted into the sequenceIndex for that key. Second, the index of the startSymbol in the global stack (the index is 0) and the queryPos (the queryPos is 0) are inserted into the queryStack, thus the tuple (0,0) is pushed to the Q_5 and Q_6 queryStack. Lastly, the queryPos of Q_5 and Q_6 is incremented by 1. The algorithm proceeds to process XML nodes (Sb,1,1) and (Sc,2,2) in Figures 4.4(b) and 4.4(c). In Figure 4.4(d), the ProcessEndSymbol(.) is invoked for Ec, and the sequenceIndex contains Q_5 and Q_6 for key Ec. The top ordered pair in the Q_5 and Q_6 queryStack is (2,2). The first component of the pair is used to retrieve the xmlParAncNode in the global stack, thus, (Sc, 2,2) is retrieved. The preorder of the current endSymbol and xmlParAncNode match, thus, the search for both Q_5 and Q_6 is advanced. The queryIds, Q_5 and Q_6, are deleted from the list of Sc and Ec in the sequenceIndex. The next symbol in the sequence of Q_5 and Q_6 is Sg and Eb, respectively. The queryIds are inserted into the sequenceIndex for their corresponding keys. For both Q_5 and Q_6,

the top entry is popped off the queryStack and the queryPos is incremented by 1. In Figure 4.4(e), the ProcessEndSymbol(.) is invoked with endlabel Eb. Q_6 is retrieved and the preorder check is verified, thus Q_6s search is advanced. First, the top element is popped off Q_6s queryStack. Second, Q_6s queryPos is incremented by 1. Lastly, Q_6 is deleted from the sequenceIndex list for keys Sc and Ec. Note that backtracking of Q_5 occurs as well. After processing Q_6, the sequenceIndex is probed for the corresponding start-symbol Sb, and Q_5 is retrieved. The preorder check returns a match, thus, indicating that Q_5 should be backtracked. Thus, the Q1 queryStack is popped, queryId Q_5 is deleted from the sequenceIndex list for key Sg, and the Q_5s queryPos is backtracked to 1. In Figure 4.4(f i) the algorithm proceeds as described by Algorithm 4.4. In Figure 4.4(j), the last query sequence symbol of Q_6 is matched and thus Q_6 is reported as a match. Q_5, however, is not reported as match because the end of the query sequence is not reached. Please note (Sd,3,2), (Ed,3,2), (Sk,7,2) and (Ek,7,2) are not shown in the Figure 4.4 since the state of the structures does not change.

Example 4.4 shows the execution of the basic filtering algorithm for non-recursive XML document. If recursion occurs in the XML documents, multiple subsequence searches may be initiated over each query sequence. The data structures and the algorithm steps were slightly modified in the final implementation of the algorithm to process multiple searches for each query sequence. For each query search initiated, a queryStack and queryPos is maintained as before. The entries inserted into the sequenceIndex are extended to include the search ID, thus, for each key the sequenceIndex will contain a list of ordered pairs that contains the queryId and searchId. The ProcessStartSymbol(.) was

slightly modified to initiate a new search when a recursive start-symbol is encountered. The filtering algorithm will first verify that the level-consistent property holds. If the level-consistent property is satisfied, then a new search is initiated and its corresponding queryStack and queryPos are updated to denote the current position of the search in the query sequence. The other operations of the filtering algorithm are maintained with the exception that multiple searches must be initiated as recursive elements are encountered.

In the second part of this section, an efficient and simple to implement algorithm is presented for the two streaming problems, namely filtering and tuple-extraction, called Forward Match. This algorithm differs from NETS-Filter since it applied dynamic programming and handles nested recursion inherently. Algorithm 4.2 provides the detailed steps of the Forward-Match algorithm. The XML document is received in a streaming fashion and parsed by a SAX parser. The NETS representation of the XML document is generated as the document is parsed, and for each 'start-symbol' or 'end-symbol' in NETS_T, Procedure computeRB is called to compute a new column of matrices R and B. Note, only the *previous* column is required to compute the next column, thus storing the entire matrix is not necessary. A match occurs if a given cell R[m][j] is equal to $|NETS_Q|$ and B[m][j] contains a bitset equal to Z, where j $\leq$ n. For the tuple-extraction problem, the values of the extraction nodes (i.e., the values of the matched leaves) must be reported for each match of Q in T. This is performed by recording the value of an extraction node on a 'start-symbol' match and associating the value with each bitset b in B[i][j], denoted by E(b). Procedure 1 computes R[i][j] per Recurrence Relation 4.1, and invokes Procedure computeBitSet to verify the level-consistent and preorder-consistent proper-

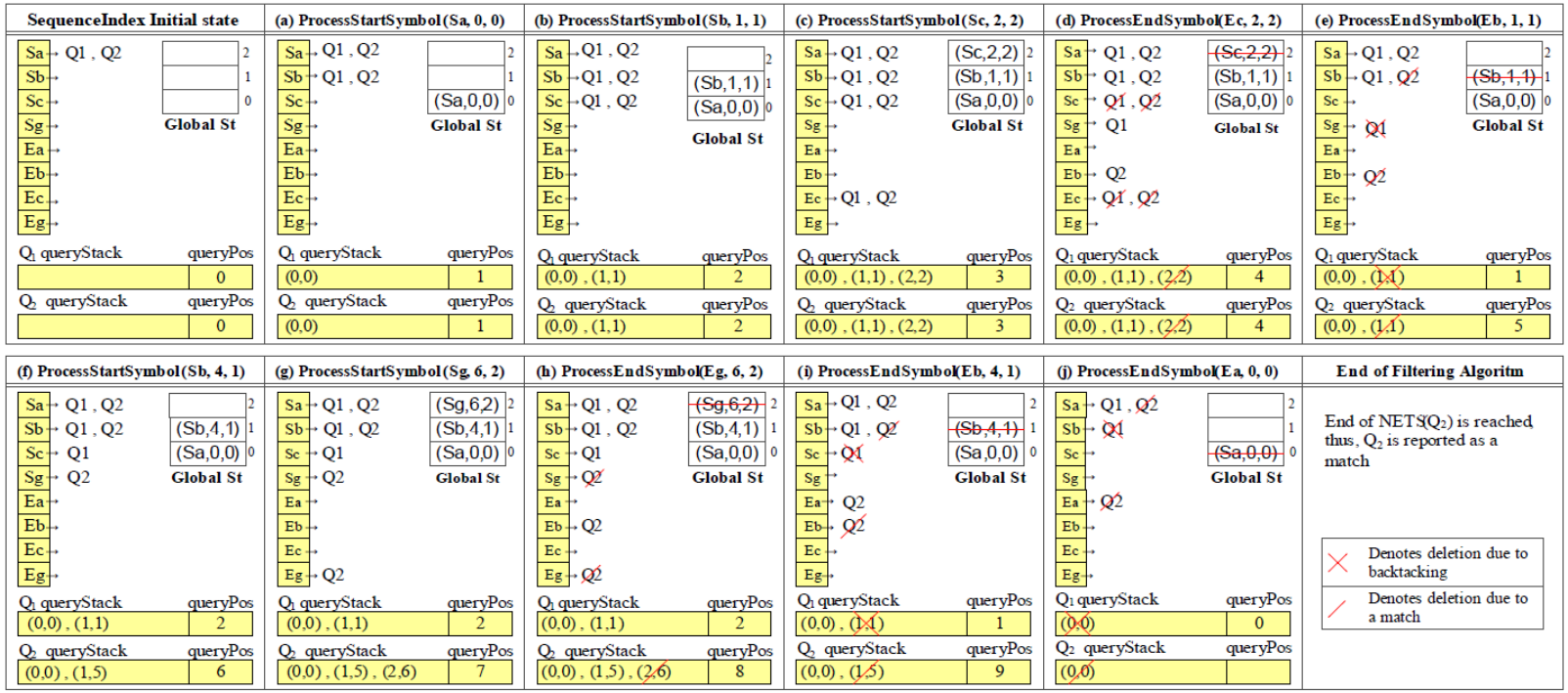


Figure 4.7: NETS-Filter Algorithm Example

ties and compute entry B[i][j] per Recurrence Relation 4.2. The level-consistent property is verified when processing a 'start-symbol'. Procedure isLevelConsistent (isLevelConsistent(.)) is called for every b in B[i][j] to verify that adding symbol $NETS_T$[j] to the subsequence associated with b will not violate the level-consistent property with respect to Q. The start-symbol in $NETS_T$[1 ... j-1] which corresponds to the parent or ancestor of $NETS_T$[j] is found by locating the most significant bit in b and then using the *preorderIndex* to lookup the location (or index) of the start-symbol in $NETS_T$. The preorderIndex table holds the position of the start and end symbols corresponding to the node's preorder number and this table is maintained by Algorithm 4.2. When considering the filtering problem, Algorithm 4.2 terminates once the end of the XML stream is reached or on a match occurrence. Algorithm 4.2 was presented for a XML document and a single query. Usually, a large collection of user profiles must be considered addressing when addressing the XML filtering problem. Sequence indexing techniques, such as the ones utilized in [41, 73], can be adopted independently of the matching algorithm. When considering the tuple-extraction problem, the entire XML stream must be processed since the objective is to report all matches as they occur. Thus, lines (34-41) of Algorithm 4.2 are performed to print the values of the extraction nodes.

4.5 Query Processing

In this section, an algorithm is presented for the query processing problem. The algorithm presented considers query matching between a query and a single XML

sequence. To generalize the algorithm for a collection of XML documents, a simple inverted index is built on the sequence labels similar to the index proposed by [78].

For the query processing problem, the two phases of query matching will be performed in sequence. First, subsequence matching is performed to compute matrix R of NETS_Q and NETS_T per Recurrence Relation 4.1. If the computed LCS length, given by entry R[m][n], is equal to the length of the query NETS sequence, then NETS_Q is said to be a subsequence of NETS_T. Given that NETS_Q is a subsequence of NETS_T from the previous step, to find all subsequence matching, backtracking over matrix R may be applied starting at the bottom-right entry R[m][n] to perform structural matching and progressively construct the result set. This approach is similar to the one utilized by LCS-TRIM for Prufer sequence matching with optimizations to reduce the number of recursive calls[78].

The backtracking approach has been evaluated in the LCS literature and noted as not computationally feasible for large sequences. For this reason, to find all LCS matches, Greenberg [33] and Rick [67] consider a graph representation to reduce the number of computations. The focus of LCS literature is to generate all LCS matches which may include those LCS which are not equal to the query sequence. While our goal is to generate all matches which are equal to the query. Thus, a new compact graph generation procedure is presented to represent all potential subsequence matches who length is equal to the query length.

In the first phase of the algorithm, graph G is generated such that all paths from initial to terminal vertices represent the set of all potential matches. In the second phase, backtracking is applied over the graph (rather than the matrix) to compute true matches

(by computing the corresponding subsequence bitsets and verifying the sequences satisfy the level-consistent and preorder-consistent properties) and retrieve the result set. The vertices and edges of G are generated as described by the GraphGeneration(.) procedure. The vertices of G are partitioned into m sets, L_1 to L_m, such that L_i= {(i,j): where 1 $\leq$ j $\leq$ n and $NETS_T$[j]=$NETS_Q$[i] and R[i][j] = i}. The edges of G are generated while computing R[i][j]. Once the graph is generated, a graph reduction procedure is applied to eliminate those vertices and edges that do not contribute to potential matches. The following is repeated until no further elimination is possible:

1. Eliminate all isolated vertices in L_i (vertices which are not connected to other vertices).

2. Eliminate all initial vertices and their edges which are not in L_m. An initial vertex is one that does not have incoming edges.

3. Eliminate all terminal vertices and their edges which are not in L_1. A terminal vertex is one that does not have outgoing edges.

The procedure EnumerateMatches(.) is called for each initial vertex w of graph G, to traverse all paths of G from initial vertices to terminal vertices. While traversing the graph, the procedure checks whether each path represents a true match by computing its corresponding bitset b and verifying whether the level-consistent and preorder-consistent properties hold. Observe that the EnumerateMatches(.) procedure follows a slightly different approach than the computeBitSet(.) procedure presented in Section 4. First, a bit in bitset b is set to *1* on a end-symbol match (rather than a start-symbol match),

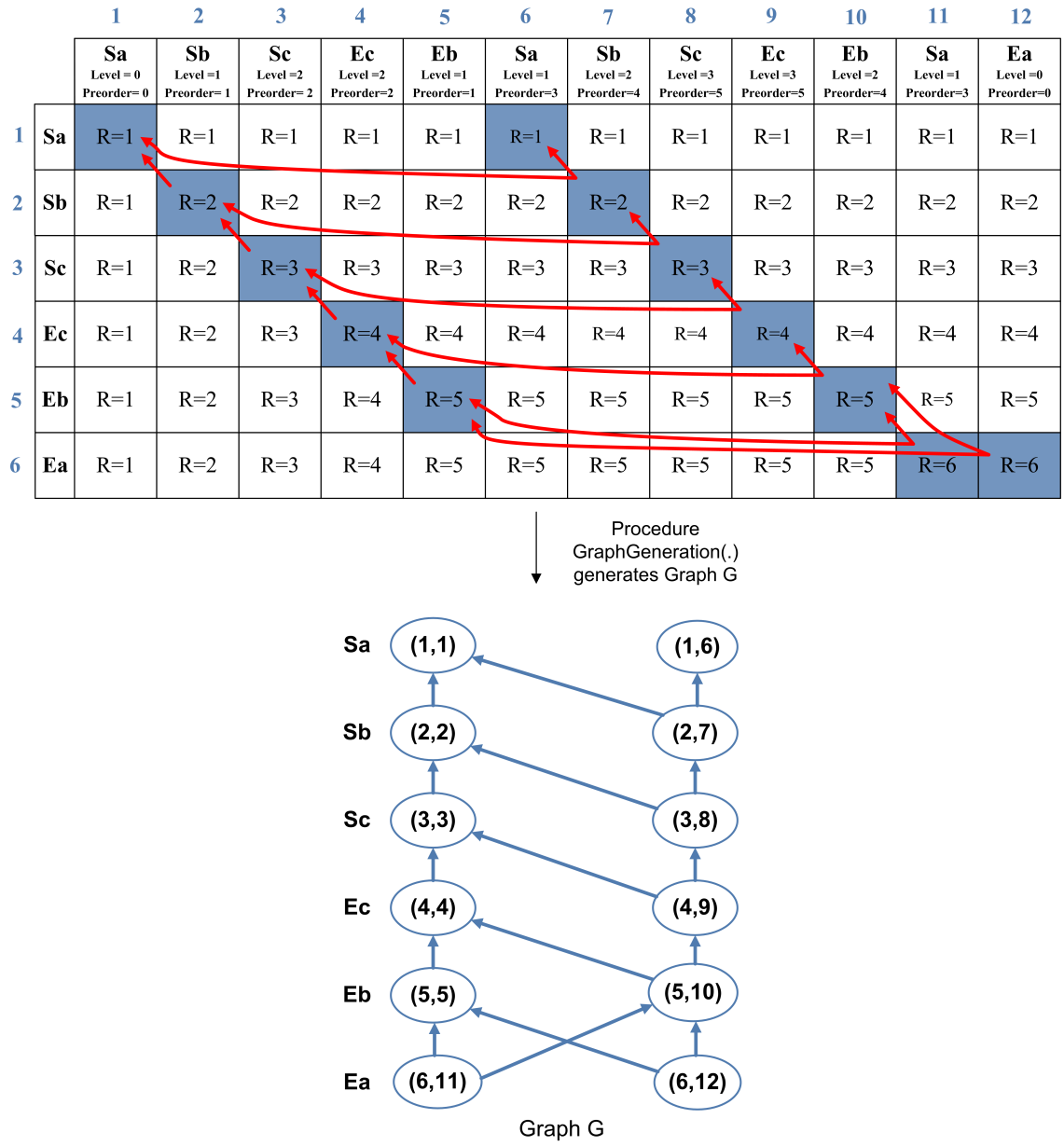


Figure 4.8: Graph G generated by the GraphGeneration(.) procedure for trees T and Q_1 in Figure 4.2.

since the sequence is processed in reverse order. The isLevelConsistent(.) procedure used in EnumerateMatches(.) is called for each vertex that represents an end-symbol to check the level-consistent property, whereas in computeBitSet(.) the procedure is called for a start-symbol since the sequence is processed in forward order. The procedure returns a match when a terminal vertex is reached and its corresponding bitset is equal to Z (all bits are set to 0). This approach is much more efficient than traditional backtracking since many irrelevant cells are skipped. Furthermore, it utilizes a graph representation of matrix R which reduces the space requirements for this approach.

Example 4.5. Consider XML tree T and Query Q_1 in Figure 4.2. Figure 4.8 shows the computed R matrix and the generated graph, G, after applying the GraphGeneration(.) procedure. The graph represents all possible subsequence matches. For this example, no vertex/edge elimination was required. The EnumerateMatches(.) procedure is called for initial vertices (6,11) and (6,12). The procedure follows each path in the graph to check whether the path represents a true match. The procedure terminates for a given path if a match occurs or when structural matching fails.

Several sequences represented by G do not result in a match. ***Path:*** **(6,11) (5,10) (4,4) (3,3) (2,2)** fails structural matching during the EnumerateMatches(.) procedure for vertex (2,2). When the EnumerateMatches(.) procedure is called for vertex (2,2), the value of the bitset b is (3,4). The procedure first checks whether the k^{th} bit in b is set to 1 (lines 32-34). This step fails since the preorder of $NETS_T[2]$=1 and the bit '1' is not set in bitset b, thus structural matching fails and the procedure terminates for this path.

The following paths are returned as matches since the terminal vertex of the path is reached and the computed bitset equals Z. The EnumerateMatches(.) procedure will return the 'Match' list, which contains the preorder of the matched nodes in the XML tree T.

4.6 Experimental Evaluation

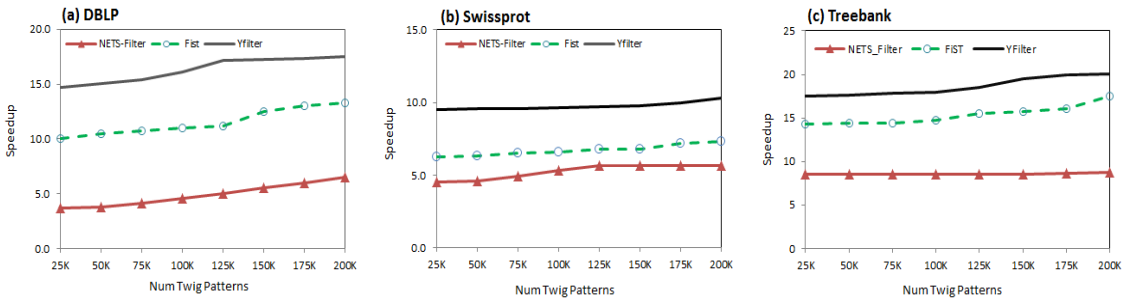


Figure 4.9: Performance comparison of Forward-Match against NETS-Filter, FiST, and YFilter for the XML filtering problem.

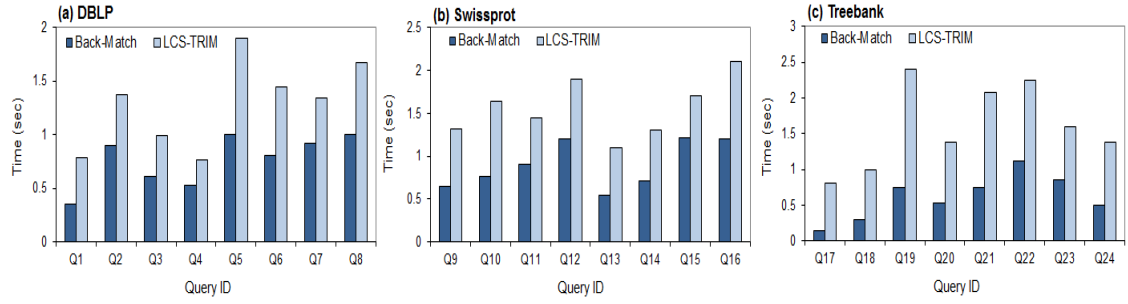


Figure 4.10: Performance comparison of Backward-Match against LCS-TRIM for the query processing problem.

This section investigates the performance of the algorithms against the best known approaches for the individual XML processing problems. All experiments were

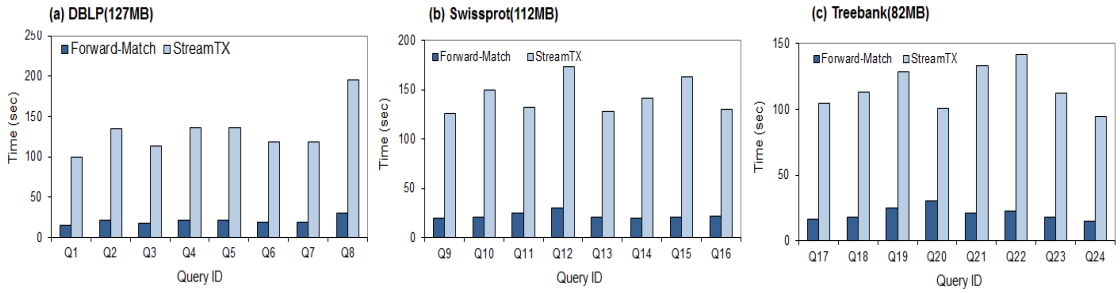


Figure 4.11: Performance comparison of Forward-Match against StreamTX for the tuple-extraction problem.

performed on a Intel Core 2 Duo 1.83Hz processor with 2GB of memory running Windows 7.

Datasets: Three XML data sets are considered: Swissprot (protein sequence), DBLP (bibliographic proceeding), and Treebank (tagged English sentences) [1]. These three datasets were chosen because they represent different domains and exhibit different characteristics. Swissport and DBLP datasets tend to have very bushy tree structure with an average depth of 5-6 nodes, while the Treebank dataset tends to have less branching but contain very deep and recursive subtrees.

Query workload: For the filtering problem, the query workload was generated for each dataset using the YFilter XPath generator [26]. The parameters used to generate twig patterns for the filtering problem were varied as follows: Num twig patterns = 25K-200K, prob. of '/' and '*' occurrence = 5% - 20%, maximum twig depth = 6 - 10, number of branches = 2 - 5. The query workload for the tuple-extraction and query processing problem is shown in Table 4.3. Most of the queries were derived from the workload proposed in [78].

[1] www.cs.washington.edu/research/xmldatasets/

Filtering Problem: First the XML filtering problem is considered, and the performance of the Forward-Match algorithm and NETS-Filter [73] is evaluated against FiST [41] and YFilter[26]. The YFilter implementation was provided by the authors of [26], while FiST was implemented based on the algorithm provided in [41], respectively. All algorithms were implemented in Java and utilized the Xerces toolkit supporting SAX 1.0 parser [2]. The approaches were run using Eclipse IDE with Java virtual machine version 1.5.0. The performance of the four approaches is evaluated for documents of various size and twig patterns with varying parameters. Figure 4.9 shows the average speedup achieved by Forward-Match over the three algorithms for 500KB documents. The filtering time includes the XML document parsing time plus twig matching. NETS-Filter is based on a stack-based approach which initiates a new subsequence search for each new subsequence match. For recursive datasets like Treebank, the performance degrades, since a new search is initiated for each recursive tag encountered. YFilter decomposes twig patterns into paths, and thus requires an expensive post-processing phase which degrades its performance for large documents. FiST represents the twig queries using Prufer sequence representation, then performs matching between the twig sequence and XML stream. FiST essentially finds every 'subsequence' match between a given query and the XML stream, records the matched nodes (preorder), then applies a post-processing phase to eliminate false positives. Forward-Match on the other hand, computes the matches in one scan of the XML stream and does not require a post-processing phase. By utilizing both the R and B matrices, false matches are pruned on the fly.

[2] www.saxproject.org/

Tuple-extraction Problem: For the stream querying (tuple-extraction) problem, the performance of Forward-Match algorithm is evaluated against the multi-tuple extraction algorithm StreamTX [36]. StreamTX is based on the Twig-Stack algorithm [13] but is modified and optimized for a streaming environment. Both Forward-Math and StreamTX algorithms were implemented in Java. Figure 4.11 shows the performance comparison between the two algorithms. Forward-Match consistently performs better than StreamTX. This performance improvement shown in Forward-Match is attributed to the fact that StreamTX must 'hold' for data until the 'closing' tag of an element is received.

Query Processing Problem: The query processing algorithm is evaluated against LCS-TRIM [78]. The Backward-Match algorithm was implemented in C++, and a C++ implementation of the LCS-TRIM provided by the authors of [78] was utlized. The performance of Backward-Match and LCS-TRIM was measured in terms of wall clock time. Both approaches were compiled and run under Cygwin environment. Approximately 300,000 documents were generated for each of the datasets, ranging in size from 100KB to 500MB. Figure 4.10 shows the comparison between Backward-Match and LCS-TRIM. The performance improvement achieved over LCS-TRIM is attributed to three reasons. First, our approach uses NETS representation instead of Prufer, thus many false matches are pruned during label subsequence matching. Second, structural matching and level checking performed by LCS-TRIM is more expensive. For example, inorder for LCS-TRIM to verify if an '//' relationship exists between two nodes, a search is initiated from the current location in the sequence to the beginning of the sequence. For queries with high occurrence of '//', the level checking will degrade the performance

of LCS-TRIM. However, for our approach, level checking has only a slight impact on the performance. Third, LCS-TRIM applied a *naive* backtracking approach over the R matrix, whereas, the Backward-Match algorithm computes a graph representation of the matches then traverses the graph to determine true matches, thus it avoids storing and visiting entries that are not part of the result.

4.7 Final Remarks

In this chapter, unified approach for *ordered* twig matching for the three XML structural processing problems was presented. The XML structural matching problem is decomposed into two sub-problems, *subsequence* and *structural* matching, which can be performed concurrently or in sequence. An algorithm, Forward-Match, for the XML filtering and tuple-extraction problems is presented. Forward-Match performs subsequence and structural matching concurrently. An algorithm is presented for the query processing problem called *Backward-Match*. Backward-Match performs subsequence and structural matching sequentially and utilizes a graph representation of all potential subsequence matches. The experiments showed that our unified approach provided significant improvement over the state-of-the art approaches for the three XML structural processing problems.

Algorithm 4.1: NETS-Filter Algorithm

```
1  procedure ProcessStartSymbol( startSymbol, preorder, level )
2  currentList = sequenceIndex[startSymbol];
3  foreach q in currentList do
4  |  xmlParAncNode = globalStack.get(indexParAnc);
   |                  /* verify level-consistent property holds */ if
   |  xmlParAncNode.levellevel satisfies querys relationship attribute then
   |  |           /* let nextSymbol denote the next symbol in query
   |  |  sequence */ sequenceIndex[nextSymbol].insert(qs queryId);
5  |  end
   |   /* let index denote the index of the parameter startSymbol
   |  in the globalStack */ push index & queryPos onto queryStack;
6  |  queryPos + = 1;
7  end
8  end procedure
9  procedure ProcessEndSymbol( endSymbol, preorder, level)
10 currentList = sequenceIndex[endSymbol];
11 foreach q in currentList do
12 |  xmlParAncNode = globalStack . get(indexParAnc);
13 |  if xmlParAncNode.preorder = preorder then
14 |  |  sequenceIndex[endSymbol].remove(qs queryId);
15 |  |  sequenceIndex[startSymbol].remove(qs queryId);
16 |  |  queryStack.pop();
   |  |   /* pop the top element off the queryStack  */ queryPos +
   |  |  = 1;
   |  |           /* let nextSymbol denote the next symbol in query
   |  |  sequence *// / if nextSymbol is null then then
17 |  |  |  report query as a match;
18 |  |  else
19 |  |  |  sequenceIndex[nextSymbol].insert(qs queryId);
20 |  |  end
21 |  end
22 end
23 currentList = sequenceIndex[startSymbol];
24 foreach  q in currentList do
25 |  xmlParAncNode = globalStack.get(indexParAnc);
26 |  if xmlParAncNode.preorder = preorder then
27 |  |  queryStack.pop();
28 |  |  delete last inserted queryId in sequenceIndex;
29 |  |  update queryPos;
30 |  end
31 end
32 end procedure
```

Algorithm 4.2: : Forward-Match - Filtering/Tuple-Extraction Algorithm

```
    Input:
    int level = -1, preorder = -1;
    int j = 1; /*index of start or end symbols of NETS_T */
    Stack globalStack;
    Output:
    list of matched queries
 1  procedure startElement(tag)        /* Start Tag Handler  */ ;
 2  NETS_T[j] = 'S' + tag ;
 3  level(NETS_T[j]) = level++ ;
 4  preorder(NETS_T[j]) = preorder++ ;
 5  preorderIndex[preorder] = j ;
 6  globalStack.push(preorder) ;
 7  processSymbol(j) ;
 8  j++;
 9  end procedure procedure endElement(tag) /* End Tag Handler */ ;
10  NETS_T[j] = 'E' + tag ;
11  level(NETS_T[j]) = level ;
12  preorder(NETS_T[j]) = pop globalStack ;
13  processSymbol(j) ;
14  level– ;
15  j++ ;
16  end procedure
17  procedure processSymbol(j) foreach i = 1 to m do
18  |  computeR&B(i,j);
19  end
                                    /* For the Filtering Problem */
20  if R[m][j] = |NETS_Q| & B[m][j] contains a bitset = Z then
21  |  report match                          /* output of filtering  */
22  end
                                /* For the Tuple-Extraction Problem */
23  if R[m][j] = |NETS_Q| then
24  |  foreach b in B[m][j] do
25  |  |  if b = Z then
26  |  |  |  print E(b);
    |  |  |                          /* output of tuple-extraction  */
27  |  |  end
28  |  end
29  end
30  end procedure
```

Procedure computeRB

```
1  compute R[i][j] per Recurrence Relation 4.1;
2  if R[i][j] < i then
3  |  B[i][j] = ∅ ;
4  else
5  |  if NETS_Q[i] = NETS_T[j] then
6  |  |  computeBitSet(i, j) ;
7  |  end
8  |  if NETS_Q[i] ≠ NETS_T[j]  OR R[i][j-1] = R[i][j] then
   |  |                    /* copy tuples from B[i][j-1] to B[i][j] */
9  |  |  foreach b in B[i][j-1] do
   |  |  |                      /* for the tuple-extraction problem  */
10 |  |  |  add 2-tuple (b, E(b)) to B[i][j] ;
   |  |  |                                  /* for the filtering problem  */
11 |  |  |  add 1-tuple (b) to B[i][j] ;
12 |  |  end
13 |  end
14 end
```

Procedure computeBitSet

```
1  foreach BitSet b in B[i-1][j-1] do
2      if NETS_T[j] is a start-symbol then
3          if isLevelConsistent(i,j,b) then
4              k =preorder(NETS_T[j]);
5              set k^th bit in b to 1;
                                  /* for the tuple-extraction problem  */
6              if NETS_Q[i] is a extraction-node then
7                  E(b) = E(b) UNION value of NETS_T[j];
8              end
9              add 2-tuple (b, E(b)) to B[i][j];
                                          /* for the filtering problem  */
10             add 1-tuple (b) to B[i][j];
11         end
12     else
               /* Verify preorder-consistent property holds  */ k =
           preorder(NETS_T[j]) if k^th bit in b equals 1 then
13             set k^th bit in b to 0 ;
                                   /* for the tuple-extraction problem */
14             add 2-tuple(b, E(b)) to B[i][j] ;
                                          /* for the filtering problem  */
15             add 1-tuple(b) to B[i][j];
16         end
17     end
18 end
```

Procedure isLevelConsistent

```
     /* most significant bit in b provides the preorder of parent
   or ancestor node  */
1  ParAncBit = most significant bit in b;
2  jParAnc = preorderIndex[ParAncBit];
3  level = level(NETS_T[j]) - level(NETS_T[jParAnc]);
4  if level satisfies NETS_Q[i] relationship attribute then
5      return true;
6  else
7      return false;
8  end
```

Procedure GraphGeneration Procedure

Input:
matrix R
Output:
Graph G
set L_1,L_2, ..., L_m to be empty
/ Each L_i for $1 \leq i \leq m$ is defined as: L_i = ((i,j): where $1 \leq j \leq n$ and $NETS_T[j] = NETS_Q[i]$ and $R[i][j] = i$) */*

```
1  foreach j = 1 to n do
2      foreach i = 1 to m do
3          compute R[i][j] per recurrence relation 4.1 ;
4          if R[i][j] = i & NETS_Q[i] = NETS_T[j] then
5              create vertex n = (i,j) ;
6              add n to L_i ;
7              foreach k = 1 to |L_{i-1}| and IF L_{i-1}(k) ≠ (*,j) do
8                  create edge (n, L_{i-1}(k)) ;
9              end
10         end
11     end
12 end
```

Procedure : GraphEnumeration(G, w, Match, b)

Input:
Graph G
Output:
List of matches

```
1  /*call procedure for each initial vertex w in G, where w=(i,j)*/ ;
2  if NETS_T[j] is a end-symbol then
3  |  if isLevelConsistent(i,j,b) then
4  |  |  k = preorder(NETS_T[j]) ;
5  |  |  set k^th bit in b to 1 ;
6  |  |  add preorder(NETS_T[j]) to Match ;
7  |  else
8  |  |  return /*terminate path match */ ;
9  |  end
10 else
11 |  /*NETS_T[j] is a start-symbol */ k = preorder(NETS_T[j]) ;
12 |  if k^th bit in b = 0 then
13 |  |  set k^th bit in b to 0 ;
14 |  else
15 |  |  return /*terminate path match */
16 |  end
17 end
18 /*check if path matched */ ;
19 if w is a leaf then
20 |  /*i.e. node is in L_1 */ if b equal Z then
21 |  |  return Match ;
22 |  else
23 |  |  return /*terminate path match */ ;
24 |  end
25 else
26 |  foreach every edge (w,x) of G do
27 |  |  EnumerateMatches(G,x,Match,b) ;
28 |  end
29 end
```

Path 1: (6,12)(5,10)(4, 9)(3, 8)(2, 7)(1, 1)
Preorder # of matched nodes of Path 1: (0,4,5)
Path 2: (6,12)(5, 5)(4, 4)(3, 3)(2, 2)(1, 1)
Preorder # of matched nodes of Path 2: (0,1,2)
Path 3: (6,11)(5,10)(4, 9)(3, 8)(2, 7)(1, 6)
Preorder # of matched nodes of Path3: (3,4,5)

Table 4.3: Query workload for DBLP (1-8), Swissprot (9-16), and Treebank(17-24)

Q_1: //article/author='Antonin Guttman'
Q_2: //phdthesis[year][series]/number
Q_3: //phdthesis[year]/number
Q_4: //inproceedings/author='E. F. Codd'
Q_5: //book[//AA93][//AABM82][//AB87a][//AB87b]//AB91
Q_6: //article[year='1999']
Q_7: //article[year='1997']/cdrom
Q_8: /article[year='1997'][volume='1']
Q_9://Entry[PFAM[@prim id=PF00304]][//SIGNAL/Descr]
Q_{10}://Entry[Org][PFAM[@prim id=PF00304]][//SIGNAL/Descr]
Q_{11}://Ref/Author='Moss J'
Q_{12}://Entry [Species][Organe='Chloroplast'] [Org='Glycine']
Q_{13}://Features/DOMAIN[from='165'][to='171'] [Descr='POLYPRO']
Q_{14}://Features[/*/from][//'171'][//'POLY-PRO']
Q_{15}://Features[//'165'][/*/to][//'POLY-PRO']
Q_{16}://Entry[/*/'Eukaryota'][/*/'Metazoa'][/*/'Craniata']
Q_{17}: //EMPTY/*/LS_OR_JJ
Q_{18}: //S/*/RB_OR_
Q_{19}: //S/SBARQ-1
Q_{20}: //NP/ADJP/IN_OR_RB
Q_{21}: //S[PRT][NP]
Q_{22}: //EMPTY//X/VP/PP/NP/S/VP/VP/NP
Q_{23}: //EMPTY[/*/NP][/*/X[//VBN][//WRB][//S[/*/ NONE]]]
Q_{24}: //VP//NP//NN

Chapter 5

Conclusions

This dissertation presented several problems and challenges dealing with data integration systems query answering over distributed set of sources or centralized database.

In the first chapter, an Online Answering System for Integrated Sources was presented. OASIS orders source accesses such that the rate at which answers are retrieved is maximized. The system considers several parameters in ordering the sources, including source coverage, overlap between sources, and cost. It addresses several challenges in building such a system, including incrementally generating source overlap estimates, ordering data sources, and selecting statistics. Future work includes extending this system to consider quality measures in query answering, such as data freshness, data accuracy, and dependency between data sources.

The second section of the dissertation focused on query answering in a single data base scenario. The second chapter of the dissertation addressed two optimization problems, ordering and selection of candidate documents for on-line query answering. For these problems, the objective functions were presented and expressed in terms of two

parameters, the probability of a query having a match in a document and the document processing time. Experimental results further validate the proposed models for estimating the two parameters. Polynomial time algorithms were presented for the document ordering problem with and without precedence constraints. The algorithms were proven to provide optimal ordering which minimizes the expected time to the first matched document. Furthermore, experimental results showed that in practice the orderings generated by the algorithms also minimize the expected time to the first k matched document, where k is a value much less than the total number of matches. The document selection problem was investigated which deals with finding a subset of documents that maximizes the expected number of matched documents for a given upper bound on total processing time. Since the document selection problem is NP-complete, a heuristic algorithm was devised. Empirical results have shown that this algorithm is effective heuristic.

While Chapter 2 presented a candidate document ordering strategy for query answering in a single source database, the last chapter addresses the challenges of query answering in such a system. The last chapter of the dissertation presented a unified approach for *ordered* twig matching for the three XML structural processing problems was presented. The XML structural matching problem is decomposed into two sub-problems, *subsequence* and *structural* matching, which can be performed concurrently or in sequence. An algorithm, Forward-Match, for the XML filtering and tuple-extraction problems is presented. Forward-Match performs subsequence and structural matching concurrently. An algorithm is presented for the query processing problem called *Backward-Match*. Backward-Match performs subsequence and structural matching sequentially and utilizes a graph representation of all potential subsequence matches. The experiments

showed that our unified approach provided significant improvement over the state-of-the art approaches for the three XML structural processing problems.

The problems addressed in this dissertation cover a spectrum of challenges faced by centralized and distributed databases. The first part of the dissertation focused on query answering in a distributed setting, while the second part of the dissertation focused on a centralized databased. These two components can be joined to build a complete query answering system for an XML-based relational or native database.

Bibliography

[1]

[2] Marklogic, http://www.marklogic.com/.

[3] Music xml.

[4] Xmark - an xml benchmark project,www.xml-benchmark.org/.

[5] Efficient filtering of xml documents with xpath expressions. In *ICDE*, pages 235–, Washington, DC, USA, 2002. IEEE Computer Society.

[6] Deepak Agarwal, Evgeniy Gabrilovich, Robert Hall, Vanja Josifovski, and Rajiv Khanna. Translating relevance scores to probabilities for contextual advertising. In *CIKM*, pages 1899–1902, New York, NY, USA, 2009. ACM.

[7] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A primitive for efficient xml query pattern matching. *ICDE*, 0:0141, 2002.

[8] Mehmet Altinel and Michael J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *VLDB*, pages 53–64, 2000.

[9] Benjamin Arai, Gautam Das, Dimitrios Gunopulos, Vagelis Hristidis, and Nick Koudas. An access cost-aware approach for object retrieval over multiple sources. *Proc. VLDB Endow.*, 3:1125–1136, September 2010.

[10] Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. On the memory requirements of xpath evaluation over xml streams. *J. Comput. Syst. Sci.*, 73(3):391–441, 2007.

[11] Jens Bleiholder, Samir Khuller, Felix Naumann, Louiqa Raschid, and Yao Wu. Query planning in the presence of overlapping sources. In *Proceedings of the International Conference on Extending Database Technology*, EDBT'06, pages 811–828. Springer, 2006.

[12] Jan-Marco Bremer and Michael Gertz. Integrating document and data retrieval based on xml. *The VLDB Journal*, 15:53–83, January 2006.

[13] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD*, pages 310–321, 2002.

[14] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD '02*, pages 310–321, New York, NY, USA, 2002. ACM.

[15] K. Selçuk Candan, Wang-Pin Hsiung, Songting Chen, Junichi Tatemura, and Divyakant Agrawal. Afilter: adaptable xml filtering with prefix-caching suffix-clustering. In *VLDB*, pages 559–570. VLDB Endowment, 2006.

[16] David Carmel, Nadav Efraty, Gad M. L, Yoelle S. Maarek, and Yosi Mass. An extension of the vector space model for querying xml documents via xml fragments. In *SIGIR*, 2002.

[17] David Carmel, Yoelle S. Maarek, Matan Mandelbrod, Yosi Mass, and Aya Soffer. Searching xml documents via xml fragments. In *SIGIR '03*, pages 151–158, New York, NY, USA, 2003. ACM.

[18] Chee Yong Chan and Yuan Ni. Efficient xml data dissemination with piggybacking. In *SIGMOD*, pages 737–748, New York, NY, USA, 2007. ACM.

[19] L.J. Chen and Y. Papakonstantinou. Supporting top-k keyword search in xml databases. In *ICDE*, pages 689–700, march 2010.

[20] Yi Chen, Susan B. Davidson, and Yifeng Zheng. An efficient xpath query processor for xml streams. In *ICDE*, page 79, 2006.

[21] Yi Chen, Wei Wang, and Ziyang Liu. Keyword-based search and exploration on databases. In *ICDE*, pages 1380 –1383, april 2011.

[22] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed xml documents. In *VLDB*, pages 263–274. VLDB Endowment, 2002.

[23] Bhaumik Chokshi, Thomas Hernandez, and Subbarao Kambhampati. Relevance and overlap aware text collection selection. 2006.

[24] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. Xsearch: a semantic search engine for xml. In *VLDB*, pages 45–56. VLDB Endowment, 2003.

[25] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[26] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.

[27] A.H. Doan and A. Hatevy. Efficiently ordering query plans for data integration. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 393 –402, 2002.

[28] Oliver M. Duschka, Michael R. Genesereth, and Alon Y. Levy. Recursive query plans for data integration. *Journal of Logic Programming*, 43:2000, 1999.

[29] Daniela Florescu, Daphne Koller, and Alon Y. Levy. Using probabilistic information in data integration. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 216–225, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[30] Daniela Florescu, Donald Kossmann, and Ioana Manolescu. Integrating keyword search into xml query processing. *Comput. Netw.*, 33:119–135, June 2000.

[31] Norbert Fuhr. Xirql: An xml query language based on information retrieval concepts. *ACM Trans. Inf. Syst.*, 22:313–356, April 2004.

[32] Gang Gou and Rada Chirkova. Efficient algorithms for evaluating xpath over streams. In *SIGMOD*, pages 269–280, 2007.

[33] Ronald I Greenberg. Fast and simple computation of all longest common subsequences. page 8, 2002.

[34] Ronald I. Greenberg. Fast and simple computation of all longest common subsequences. *CoRR*, cs.DS/0211001, 2002.

[35] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. Xrank: ranked keyword search over xml documents. In *SIGMOD*, pages 16–27, New York, NY, USA, 2003. ACM.

[36] Wook-Shin Han, Haifeng Jiang, Howard Ho, and Quanzhong Li. Streamtx: extracting tuples from streaming xml data. *Proc. VLDB Endow.*, 1(1):289–300, 2008.

[37] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40:11:1–11:58, October 2008.

[38] Zachary Ives, Daniela Florescu, Inria Roquencourt, Marc Friedman, Alon Levy, and Daniel Weld. An adaptive query execution system for data integration. In *Sigmod*, pages 299–310, 1999.

[39] Vanja Josifovski, Marcus Fontoura, and Attila Barta. Querying xml streams. *The VLDB Journal*, 14(2):197–210, 2005.

[40] Raghav Kaushik, Christopher Ré, and Dan Suciu. General database statistics using entropy maximization. In *Proceedings of the 12th International Symposium on Database Programming Languages*, DBPL '09, pages 84–99, Berlin, Heidelberg, 2009. Springer-Verlag.

[41] Joonho Kwon, Praveen Rao, Bongki Moon, and Sukho Lee. Fist: scalable xml document filtering by sequencing twig patterns. In *VLDB*, pages 217–228. VLDB Endowment, 2005.

[42] Joonho Kwon, Praveen Rao, Bongki Moon, and Sukho Lee. Value-based predicate filtering of xml documents. *Data Knowl. Eng.*, 67(1):51–73, 2008.

[43] Joonho Kwon, Praveen Rao, Bongki Moon, and Sukho Lee. Fast xml document filtering by sequencing twig patterns. *ACM Trans. Internet Technol.*, 9:13:1–13:51, October 2009.

[44] Stephen Robertson Leif Azzopardi, Gabriella Kazai. Advances in information retrieval theory: Second international. 2009.

[45] Jianxin Li, Chengfei Liu, Jeffrey Xu Yu, and Rui Zhou. Efficient top-k search across heterogeneous xml data sources. In *DASFAA*, pages 314–329, Berlin, Heidelberg, 2008. Springer-Verlag.

[46] Quanzhong Li and Bongki Moon. Indexing and querying xml data for regular path expressions. In *VLDB*, pages 361–370. Morgan Kaufmann Publishers Inc., 2001.

[47] Shaorong Liu, Qinghua Zou, and Wesley W. Chu. Configurable indexing and ranking for xml information retrieval. In *SIGIR '04, year = 2004, location = Sheffield, United Kingdom, pages = 88–95, publisher = ACM, address = New York, NY, USA, keywords = XML indexing, XML information retrieval, XML ranking, ranking,.*

[48] Xiping Liu, Changxuan Wan, and Lei Chen. Effective xml content and structure retrieval with relevance ranking. In *CIKM '09*, pages 147–156, New York, NY, USA, 2009. ACM.

[49] Ziyang Liu and Yi Chen. Processing keyword search on xml: a survey. *WWW*, pages 1–37, 2011.

[50] Jiaheng Lu, Tok Wang Ling, Tian Yu, Changqing Li, and Wei Ni. Efficient processing of ordered xml twig pattern. In Kim Viborg Andersen, John Debenham, and Roland Wagner, editors, *Database and Expert Systems Applications*, volume 3588 of *Lecture Notes in Computer Science*, pages 300–309. Springer Berlin / Heidelberg, 2005.

[51] Amelie Marian, Sihem Amer-Yahia, Nick Koudas, and Divesh Srivastava. Adaptive processing of top-k queries in xml. In *ICDE*, pages 162–173, Washington, DC, USA, 2005. IEEE Computer Society.

[52] Iris Miliaraki and Manolis Koubarakis. Distributed structural and value xml filtering. In *DEBS*, pages 2–13, New York, NY, USA, 2010. ACM.

[53] Roger Moussalli, Mariam Salloum, Walid Najjar, and Vassilis Tsotras. Accelerating xml query matching through custom stack generation on fpgas. In *High Performance and Embedded Architecture and Compilation*, HiPEAC, 2010.

[54] Roger Moussalli, Mariam Salloum, Walid Najjar, and Vassilis Tsotras. Massively parallel xml twig filtering using dynamic programming on fpgas. In *Proceedings of the 27th International Conference on Data Engineering*, ICDE 2011, 2011.

[55] Felix Naumann, Johann-Christoph Freytag, and Ulf Leser. Completeness of integrated information sources. *Inf. Syst.*, 29:583–615, September 2004.

[56] Zaiqing Nie, Subbarao Kambhampati, and Ullas Nambiar. Effectively mining and using coverage and overlap statistics for data integration. *IEEE Trans. on Knowl. and Data Eng.*, 17:638–651, May 2005.

[57] Zaiqing Nie, Subbarao Kambhampati, Ullas Nambiar, and Sreelakshmi Vaddi. Mining source coverage statistics for data integration. In *Proceedings of the Web Information and Data Management*, WIDM'01, 2001.

[58] Zaiqing Nie, Ullas Nambiar, Sreelakshmi Vaddi, and Subbarao Kambhampati. Mining coverage statistics for websource selection in a mediator. In *Proceedings of the International Conference on Knowledge Management*, CIKM.

[59] Dan Olteanu. Spex: Streamed and progressive evaluation of xpath. *IEEE Transactions on Knowledge and Data Engineering*, 19:934–949, 2007.

[60] Makoto Onizuka. Processing xpath queries with forward and downward axes over xml streams. In *EDBT*, pages 27–38, New York, NY, USA, 2010. ACM.

[61] Feng Peng and Sudarshan S. Chawathe. Xsq: A streaming xpath engine. *ACM Trans. Database Syst.*, 30(2):577–623, 2005.

[62] Aneesh Raj and P. Sreenivasa Kumar. Branch sequencing based xml message broker architecture. *International Conf. on Data Engineering*, 0:656–665, 2007.

[63] P. Rao and B. Moon. Sequencing xml data and query twigs for fast pattern matching. *ACM Trans. Database Syst.*, 31:299–345, March 2006.

[64] Praveen Rao and Bongki Moon. Prix: Indexing and querying xml using prüfer sequences. page 288, 2004.

[65] Louiqa Raschid, Esther Vidal, Yao Wu, Felix Naumann, and Jens Bleiholder. Answering top k queries efficiently with overlap in sources and source paths. 2007.

[66] Christopher Ré and Dan Suciu. Understanding cardinality estimation using entropy maximization. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '10, pages 53–64, New York, NY, USA, 2010. ACM.

[67] Claus Rick. Efficient computation of all longest common subsequences. In *Algorithm Theory - SWAT 2000*, volume 1851 of *Lecture Notes in Computer Science*, pages 687–697. Springer Berlin / Heidelberg, 2000.

[68] Armin Roth. Completeness-driven query answering in peer data management systems. In *Proc. of Very Large Databases*, VLDB, Vienna, Austria, 2007.

[69] Armin Roth and Felix Naumann. Benefit and cost of query answering in pdms. In *Proceedings of the 2005/2006 international conference on Databases, information systems, and peer-to-peer computing*, DBISP2P'05/06, pages 50–61, Berlin, Heidelberg, 2007. Springer-Verlag.

[70] Armin Roth, Felix Naumann, Tobias Hubner, and Martin Schweigert. System p: Query answering in pdms under limited resources. In *WWW*, Edinburgh, UK, 2006.

[71] Mariam Salloum and Vassilis Tsotras. Efficient and scalable sequence-based xml filtering. In *12th International Workshop on the Web and Databases*, WebDB 2009, 2011.

[72] Mariam Salloum, Vassilis Tsotras, Divesh Srivastava, and Luna Dong. Optimizing candidate document selection for query answering in xml databases. In *5th International Workshop on Ranking in Databases*, DBRank 2011, 2011.

[73] Mariam Salloum and Vassilis J. Tsotras. Efficient and scalable sequence-based xml filtering. In *WebDB*, 2009.

[74] Anish Das Sarma, Xin Luna Dong, and Alon Halevy. Data integration with dependent sources. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 401–412, New York, NY, USA, 2011. ACM.

[75] Torsten Schlieder and Holger Meuss. Querying and ranking xml documents. *J. Am. Soc. Inf. Sci. Technol.*, 53:489–503, May 2002.

[76] et. al. Shurug Al-Khalifa. Structural joins: A primitive for efficient xml query pattern matching.

[77] Shirish Tatikonda, Srinivasan Parthasarathy, and Matthew Goyder. Lcs-trim: dynamic programming meets xml indexing and querying. In *VLDB '07*, pages 63–74, 2007.

[78] Shirish Tatikonda, Srinivasan Parthasarathy, and Matthew Goyder. Lcs-trim: dynamic programming meets xml indexing and querying. In *VLDB*, pages 63–74, 2007.

[79] Anja Theobald and Gerhard Weikum. The index-based xxl search engine for querying xml data with relevance ranking. In *EDBT '02*, pages 477–495, 2002.

[80] V. Vassalos and Y. Papakonstantinou. Using knowledge of redundancy for query optimization in mediators. In *AAAI Workshop on AI and Information Integration*, 1998.

[81] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[82] Haixun Wang and Xiaofeng Meng. On the sequencing of tree structures for xml indexing. In *ICDE*, pages 372–383. IEEE Computer Society, 2005.

[83] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. Vist: a dynamic index method for querying xml data by tree structures. In *SIGMOD*, pages 110–121. ACM, 2003.

[84] Felix Weigel, Holger Meuss, Klaus U. Schulz, and François Bry. Content and structure in indexing and ranking xml. In *WebDB '04*, pages 67–72, New York, NY, USA, 2004. ACM.

[85] Yu Xu and Yannis Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD*, pages 527–538, New York, NY, USA, 2005. ACM.

[86] Pavel Zezula, Giuseppe Amato, Franca Debole, and Fausto Rabitti. Tree signatures for xml querying and navigation. pages 149–163, 2003.

[87] Pavel Zezula, Federica Mandreoli, Federica M, and Riccardo Martoglia. Tree signatures and unordered xml pattern matching. 2004.