UNIFIED STRUCTURE AND CONTENT SEARCH FOR PERSONAL INFORMATION MANAGEMENT SYSTEMS

BY WEI WANG

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Amélie Marian

and approved by

New Brunswick, New Jersey

October, 2010

ABSTRACT OF THE DISSERTATION

Unified Structure and Content Search for Personal Information Management Systems

by Wei Wang

Dissertation Director: Amélie Marian

The ability to quickly retrieve files in personal information systems is becoming increasingly important as users store and collect ever larger amounts of personal data. This explosion of information is driving a critical need for complex search tools to access often very heterogeneous data in a simple and efficient manner.

Numerous search tools have been developed to locate personal information stored in file systems. They often allow for some *ranking* on the textual part of the query, but only consider structure (e.g., file directory) and metadata (e.g., date, file type) as *filtering* conditions. However, simple keyword-only searches are often insufficient. First, files that are very relevant to the keywords, but which do no satisfy the exact filtering conditions would not be considered valid answers. Second, search tools often strictly separate the directory information from content information, and relevant answers that do not adhere to this strict separation would be missed. Last, current tools do not consider relationships between structure and content, possibly across multiple directories and/or files, even if the additional information helps narrow search results and improves search accuracy.

Because of the heterogeneity of data in personal information systems, we believe it is critical to support approximate matches on both the structure and content components of queries and to allow for query conditions to be evaluated across file boundaries.

In this thesis, we developed a suite of techniques that allow search tools to effectively and efficiently evaluate flexible query conditions against directory structure and structure contained within files. Our work started by considering how to score fuzzy conditions in each query dimension (i.e., structure, content, metadata) and meaningfully combine individual dimension scores. We then proceeded to consider how to unify structure and content, such that users can specify a single query that contains both structure and content components and that can be evaluated at once across file boundaries. Finally, we have designed algorithms and data structures to support efficient processing of the multi-dimensional as well as unified queries. We evaluated our techniques and our results show that the system has the potential to significantly improve ranking accuracy and is practical for everyday usage.

Acknowledgements

Throughout the long journey of PhD, I greatly benefited from the guidance and support of many people.

First and foremost, I would like to thank my advisor, Professor Amélie Marian, whose guidance, support, and patience have allowed me to finish this work. I have learned so much – whether in doing research, writing papers or giving presentations.

I am extremely grateful to my co-advisor, Professor Thu Nguyen, for being a constant source of encouragement and inspiration. Thu, I truly appreciate the time and respect you have given to me over the years.

The work on multi-dimensional search was done with my colleague Christopher Peery. Chris, I really enjoyed every second when I worked with you. You are a true friend and always make life cheerful.

I also thank Kien Le and Tuan Phan, for your friendship and endless laughter.

I thank the staff of the Computer Science Department for always accommodating my myriad of requests with smiling graciousness.

I thank Professor Shanqing Li for all the wonderful discussions and guidance on a variety of topics. You have been an excellent role model and mentor. I have learned so much from you.

I would also like to thank my parents, my sister, and my brother in law. They were always supporting me and encouraging me with their best wishes.

Finally, I would like to thank my wife, Hongbo Liu. She was always there cheering me up and stood by me through the good times and bad.

Dedication

To my family, without whose patience and support it could never have been achieved.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Personal information management relates to the acquisition, organization, maintenance, retrieval, and usage of information items such as documents, pictures, and email messages [70]. Ideally we would like to have the right information, in the right place and in the right form to meet our needs. A large number of personal information management tools have been developed to help us work more efficiently while spending less time with time-consuming and error-prone activities. For example, numerous search tools have been developed to help us find the information we need, even when it gets buried in massive amount of only loosely related information.

Nevertheless, reality is far from the ideal for many people. Difficulties arise from several ongoing trends in computing that affect how people acquire and manage information. The first trend is the drastic increase in capacity and reduction in price of storage. It seems much cheaper to keep everything than it does to assume the overhead of up-front evaluation and discard of individual items [66]. The second trend is the increasing ubiquity of digital devices and network access, allowing people to easily garner, share, and exchange data through networks (e.g., email messages, instant messaging) and digital devices (e.g., PDA, smart phone, digital camera). Finally, increasing number of new tools are build to help people manage the ever-growing personal data. People rely on more and more tools to accomplish their tasks in daily life.

These trends present a set of challenges. First, the ubiquity of digital devices and network access is allowing (and motivating) users to gather large bodies of personal data. Second, abundant storage reduces the need to prune personal data regularly;

however, the increasing data volume makes it difficult to *track the location and memorize detailed information about data* that was accumulated over time. It also *slows down the personal information management tools* when they process data. Next, while the networks and personal devices offer the opportunities to access a variety of information, the data is often stored in *heterogeneous organizational schemes and formats*, which were not designed to constitute a coherent collection. Last, new tools, even as they become smarter and help in some areas, often *aggravate the problem of information fragmentation.* That is, the information may be nearby but it is "locked" in separate application "silos" and cannot be easily extracted.

To find information that we need in such a computing environment, there is a critical need for search tools that can help users to quickly find relevant information at will. Technically, this translate to the need for search tools that provide both high-quality scoring mechanisms and efficient query processing capabilities.

Numerous search tools have been developed to perform keyword searches and locate personal information stored in personal information management systems such as the commercial file system search tools *Google Desktop* [50] and *Spotlight* [79]. However, these tools usually index text content, allowing for some *ranking* on the textual part of the query—similar to what has been done in document search in the Information Retrieval (IR) community—but only consider structure (e.g., file directory) and metadata (e.g., date, file type) as *filtering* conditions.

Unfortunately, simple keyword-only searches plus filtering are often insufficient and they have at least three deficiencies. First, files that are very relevant to the keywords, but which do no satisfy the exact filtering conditions would not be considered as valid answers. Second, such tools often strictly separate the (external) directory information from (internal) content information, and relevant answers that do not adhere to this strict separation would be missed. Third, current tools do not consider relationships between structure and content, possibly across multiple directories and/or files, even if the additional information helps narrow search results and improves search accuracy. These deficiencies are illustrated by the following example:

Example 1 *Consider John, a user saving personal information in a file system. John wants to retrieve photos of a Halloween party that was held at his home where someone was wearing a witch costume. Unfortunately, John cannot recall the detailed information about those photos, although he vaguely remembers the photos are saved in "jpg" file format and they are located in a directory called "home" that might have a sub-directory "processed" containing polished photos.*

Ideally, the file directory structure would have been created and maintained consistently and all photos are properly tagged. In real-life scenarios, this is rarely the case: users change their file organizations over time, inconsistently annotate their data and may gather information from different sources. In our example, John has changed the way he organizes his photos over time when he switched to a new computer that has more storage space and decided to use a new photo organization software, and his pictures are not consistently tagged. As a result, pictures from Halloween parties held in different years match very different directory structures and do not necessarily have matching tags, as illustrated in Figure 1.1. In addition, some relevant pictures are not in John's main photo hierarchy, but in his email folder as they were sent to him by friends.

In this scenario, three difficulties arise for current search tools.

First, John is limited to using a keyword query "home, witch" to search photos, while his knowledge of file format (jpg) and file tags ("Halloween" in the caption tag) are used as filtering conditions. Photos such as the file party42.jpg*, which contains the keywords "witch" and "home" but does not have a caption tag with value "Halloween", would not be returned although it may be a suitable approximate match to the query.*

Second, John knows keyword "Halloween" could appear in both the textual content of the image files and the names of directories. For John's data, both cases should be considered as relevant to meet his needs. However, current search tools would not return the file pic1728.gif *since "Halloween" is expected to be a content term (i.e., a term in the tag string) and not part of the directory hierarchy.*

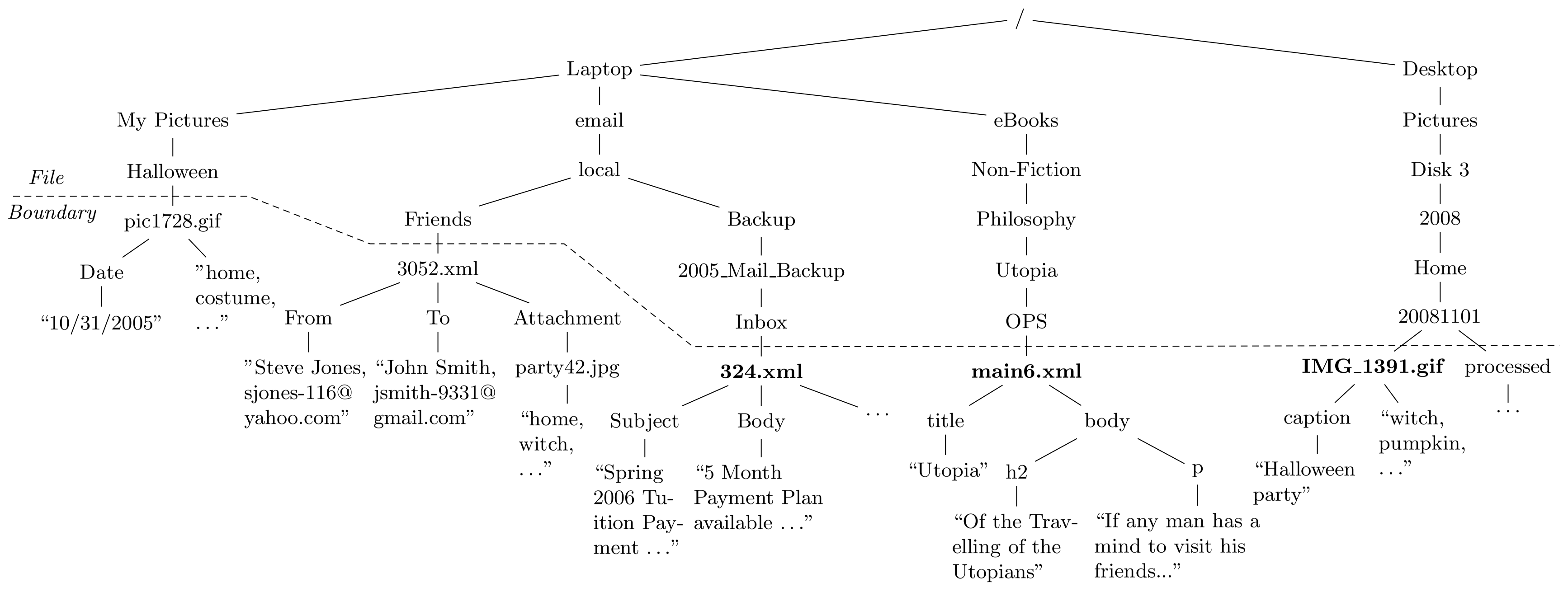


Figure 1.1: A subset of an example user's personal information file system. The user integrates multiple file systems into a single one by mounting them all on a single device.

And third, using current search tools, John could not pinpoint the file IMG_1391.gif *by providing a query containing the following correlation between the content and directory information: files that satisfy the keyword condition "Halloween, witch" and a sub-directory "processed" are under the* same *directory "home". In such scenarios, leveraging such relationships is very useful, since "home" is a popular term and many answers of various relevance could be returned. The information about sub-directory "processed" represents a context of the content information and has the potential to drastically narrow search results.*

From users' perspective, unlike searches over a vast unknown data collection like the Web, users are familiar with many different characteristics of their information. They possibly have in-depth knowledge of where they expect the file to be located (directory structure) and of structural information contained within the file (internal structure). This additional information has the potential to significantly increase the accuracy of search in personal information systems.

However, users are notoriously bad at remembering exactly where they stored a particular file or how the files are structured [30]. For example, if directory information is part of a search query, the query is likely to be incomplete or contain mistakes, as users often confuse or misremember the order of the directory components, their relationships, or their labels. The extra information such as the directory structure is nevertheless useful as it gives meaningful context information to a general content query. Therefore it is desirable to provide an approximation mechanism that leverages any accurate information in the query without penalizing mistakes too strongly.

Because of the abovementioned complexities for personal information search, we believe it is critical for search tools to possess two key capabilities. First, it cannot be restricted to keyword search but combine and approximate other information or context associated with files (e.g., directory information). And second, it should cut across application silos and file boundaries, and manipulate all personal information in a unified fashion.

1.2 Thesis Outline

In this thesis, we develop a suite of techniques that allow search tools to effectively and efficiently consider flexible query conditions along dimensions other than content. We assume that users could access his personal information via a single logical file system by using some mechanism, such as cross mounting, to integrate the file systems of his computing devices. The logical file system is viewed as a data tree that includes the textual *content* of the files, (a potentially large amount of) *metadata* information (e.g., modification time, file type), as well as some *structural* information (e.g., directory structure, internal structure of files). Internal structure can be derived from the file format, e.g., `<from>` and `<to>` fields in email files, or could consist of annotation data associated with the files, e.g., tags given to photo files.

The thesis consists of four main parts as follows.

Multi-dimensional Search

We begin by considering how to score fuzzy conditions in each query dimension (i.e., structure, content, metadata) and meaningfully combine individual dimension scores.

We argue that allowing flexible conditions on structure and metadata can significantly increase the quality and usefulness of search results in many search scenarios. The challenge is then to adequately score the search results by taking into account flexibility in the textual component *together* with some flexibility in the structural and metadata components of the query. Once an adequate scoring mechanism is chosen, efficient algorithms to identify the best query results, *without considering all the data in the system*, are also needed.

To this end, we present a novel multi-dimensional search framework that allows users to provide fuzzy conditions on three query dimensions: content, metadata, and structure. We describe individual *IDF*-based scoring approaches for each of these dimensions (scoring of the metadata dimension was done outside the context of this thesis and we only present a brief overview) and present a unified scoring framework which aggregates the individual dimensions scores to produces a single relevance score for each file. As the first step, we primarily consider the directory structure for the structure dimension. The internal file structure will be considered in our work on the unified structure and content search.

We perform an evaluation using a real-world data set. We show that our approach is able to provide relevant results for query scenarios where current search tools often fail to return answers. We demonstrate that our *IDF*-based scoring approach provides a meaningful distribution of scores that captures the specificity of each dimension. We also show that our multi-dimensional score aggregation technique preserves the properties of individual dimension scores and has the potential to significantly improve ranking accuracy.

Efficient Query Processing

We then consider the index structure and algorithms for efficient query processing.

To support fuzzy query conditions, we use query relaxation operations to relax a query to less restricted forms to permit approximate answers. Efficiently supporting the relaxed queries and scoring techniques for multi-dimensional search requires new indexing structures and algorithms designed to handle query relaxations. The structure dimension, which exhibits complex relaxations is particularly challenging because:

- The index structure for the structure dimension is query dependent. It is not practical to construct a query-independent index of the file system because this would entail enumerating all possible relaxations of all directory pathnames, making efficient query-dependent index construction a critical issue.
- The query relaxation indexes grow exponentially with the size of the query condition. The index structures and algorithms should scale to handle a possibly large number of approximate matches.

In this part of the thesis, we focus on the efficiency of our approach and present new data structures and index construction optimizations to address the above challenges. Our techniques and data structures work in conjunction with our adaptation of existing top-k algorithms to provide an efficient implementation of our overall framework. We show how these indexes can be optimized for both sorted and random accesses that are necessary to support a top-k query processing algorithm such as the Threshold Algorithm [43].

We evaluated our implementation by executing a large number of queries against a large, real-life personal data set. Our evaluation shows that our indexes and optimizations are necessary to make multi-dimensional searches efficient enough for practical everyday usage. We also show that our optimized query processing strategies exhibit good behavior across all dimensions, resulting in good overall query performance and good scalability.

While our discussion focuses on the query processing for multi-dimensional search, the same techniques can be adapted to the unified structure and content search that are discussed in the next part of this thesis.

Unified Structure and Content Search

Next, we turn to considering how to unify structure and content, such that users can specify a single query that contains both structure and content components and that can be evaluated simultaneously across file boundaries.

Earlier we have pointed out the collections of personal information are often very heterogeneous. We believe it is critical to support approximate matches on both the content and structural components of the query and to allow for query conditions to be evaluated across file boundaries. For this purpose, we use a unified data model, in the spirit of [38] to represent user data. We propose a query model that supports approximation in both the structure and content component of the queries, and allows for structure conditions to be matched by content terms and vice versa.

Unlike the previous work on multi-dimensional search which supports approximation in query conditions on three dimensions: content, metadata and structure and aggregates individual dimension scores to rank answers, we present a unified scoring framework that simultaneously considers relaxed query conditions on structure and content to provide a unified score. In addition, our techniques rely on a unified data model that ignores the traditional physical file boundaries. We allow for users to provide tree-shaped (*twig*) queries (see Figure 5.1(a) for an example) for more complex query conditions. For simplicity, we decompose twig queries into path-shaped (*path*) queries. Answers are then searched against path queries and their scores of individual path queries are aggregated to produce the final unified scores.

Our experimental evaluation shows that our unified approach improves search accuracy over existing content-based methods by leveraging information from both structure and content as well as relationships between the terms. Our work shows the importance of allowing for structural query approximation in personal information queries and opens many important open research directions for efficient and high-quality search tools.

Twig Queries

Last, we advance the unified structure and content search framework and turn to the complete tree-shaped twig queries that encapsulate all correlations between structure and content components.

As illustrated in Example 1, often users need to specify the correlations between certain query conditions. Such correlations can only be represented by twig queries and thus it is necessary to perform personal search directly base on twig queries. Further, to support flexible query conditions, the unified search framework in the previous part should also be adapted to the search for twig queries.

Due to the complexity of twig queries, in this thesis we focus on extending the query model of path queries for twig queries and discussing alternatives to approximate twig queries and score personal files. We present the extended unified structure and content scoring framework that considers relaxed twig query conditions. We leave open problems such as flexible result granularity and efficient twig query matching and processing to future work.

1.3 Contributions

The contributions of this thesis are as follows:

- In Chapter 3, we present a design and evaluation of a unified multi-dimensional search framework. This framework allows users to provide flexible query conditions on three dimensions: content, metadata, and structure.

- In Chapter 4, we present efficient index structures and optimizations to improve query processing. Our optimizations take into account the top-k algorithm evaluation strategy to focus on building only the parts of the index that are most relevant to the query processing.
- In Chapter 5, we present a unified data and query model to unify the external and internal structure of files for query conditions to cross the file boundaries. New techniques are developed to support the complete structure data in file systems and they adequately consider the new structural relationships.
- In Chapter 6, we extend the existing query model to handle complete twig queries and discuss the alternatives to approximate twig queries and score matching data.
- We implemented a prototype system based on the ideas in Chapter 3–5. Our experimental results show that the system has the potential to significantly improve ranking accuracy and it is practical for everyday usage.

Chapter 2

Background and Related Work

2.1 Models for Information Management

Dataspace [33, 48, 54] is an abstraction of data entities for information management. Data in a dataspace can be referenced by an identifier, and is linked with other data across spaces and domains. The support system for dataspace often contains a set of services such cataloging and browsing, searching and querying, updating, and monitoring services. Several personal dataspace management systems [14, 39, 41, 65, 82] have been developed based on the vision of dataspaces.

The SEMEX [19] and Haystack [60] systems have focused on the user perspective of personal information management. These works allow users to organize personal data semantically by creating associations between files or data entities. They provides a logical view of the user's personal information, based on meaningful objects and associations. These associations are then used to enhance search. While the works are aimed at helping users better organize and associate information, we focus on improving personal searches by allowing for users to specify more flexible query conditions.

Other works [38, 89] address information management by proposing generic data models for heterogeneous and evolving information. In [72] the author proposes an ordered tree model for desktop search. It also briefly discusses returning data segments at different granularity. For the data model in this work, we view the personal data as a rooted, unordered tree.

Several file system related projects have tried to enhance the quality of search within file system by leveraging the context in which information is accessed to find related information [78] or by altering the model of the file system to a more object-orientated database system [15]. These works attempt to leverage additional semantic information to locate relevant files while our focus is in determining the most relevant piece of information based solely on a user-provided query.

2.2 Personal Information Search

Recently there has been a surge in projects attempting to improve Desktop search [29, 50, 79]. These projects provide search capabilities over content and then employ other pieces of information such as size, date, or types as filtering conditions. This approach is insufficient in many search scenarios since it excludes files that are very relevant to the query but do no satisfy the exact filtering conditions, while our approach allows for all relevant candidate files to be ranked and returned based on the more flexible fuzzy query conditions.

Lucene [47] is an open source information retrieval software library. It has been widely used in the implementation of Internet search engines and local site search tools. In this work, Lucene is used for the implementation of keyword search for textual content. It is also used as a basis for the comparison with other search techniques.

Learning and query selectiveness based ranking techniques are proposed in [28] for desktop search. Their ranking formula uses linear functions to combine the weights for various file features (e.g., filename, size, date of creation). In contrast, our work is aimed at an integrated vision of the data and allows for an organic representation of query conditions across the file boundaries.

Other desktop search tools [22, 24, 26, 53, 67] exploits semantic associations of data items collected from explicit and implicit user activities, while our search techniques focus on querying the existing textual and structural information in the file system.

2.3 User Study in Desktop Environment

Research has shown that desktop users tend to prefer location-based search to a keyword-based search [10,11,40,58,63], which observes that users rely on the information such as directories, dates, and names when searching files. Another study [80] investigates user behavior when searching emails, files, and the web. Even if users know exactly what they are looking for, they often navigate to their target in small steps, using contextual information such as metadata information, instead of keyword-based search. Further, the results in [27] suggest folders (placing) should be combined with labels (tagging) as means of organizing personal information. These studies motivate the need for personal information management system search tools that use metadata and structure information.

Other study [31, 32] reveals that users are rarely allowed to complete a single task without significant amount of task switching and numerous interruptions. Memory for important computing events is often quite fragile. Our work considers users may forget details about data and allows for users to provide incomplete query conditions or queries containing various mistakes.

The user study [13] shows that location based search strategy also largely dominates the recall-directed search phase. Additionally, the most frequently free-recalled attributes are the characteristics of the documents' textual content (e.g., abstract, structure, distinctive portions of text like the title etc.). Therefore, both the external directory structure and internal file structure play important roles for the search of personal information. Furthermore, these studies conclude that systems should take into account the approximation of the recall in the returned results by allowing errors in the query parameters. Our work addresses this issue by allowing for fuzzy query conditions and relaxing query conditions automatically during search.

The results in [12] suggests that user still prefer (folder-based) navigation over search regardless of the improved desktop search engines (i.e., Google Desktop, Spotlight). This confirms that the content-centric search strategies are insufficient to meet users' need. It is our belief that a more flexible search strategy could better help users by fully utilizing metadata and structural information in addition to the textual content of files.

2.4 Query Relaxation

In Information Retrieval, queries can be relaxed through query expansion [8,73]. Search tools use query expansion to add related terms to the original query to archive better precision and recall. Typically the terms that are added are either synonyms of words (from pre-built lists such as WordNet [44, 86]) or in various morphological forms (by stemming). Spelling mistakes are also fixed and the correct forms are included in queries. Although our search framework can be complemented with query expansion techniques, in this thesis, we focus on the relaxation of structural relationships and metadata value ranges. We also develop scoring frameworks based on the query relaxation operations.

Several query relaxation strategies have been proposed in [25, 34, 49, 59, 74]. More recently, XML structural query relaxations have been discussed in [5–7]. These works focus on the relaxation operations that are suitable for XML documents: edge generalization, leaf deletion, and subtree promotion. In contrast, we are looking at personal information file system which have more variations in the formats of semi-structured data they handle. Our work uses ideas introduced in the XML context, such as the DAG indexing structure to represent all possible structural relaxations [6], or the relaxed query containment condition [6, 7]. Our techniques differ from these work as we consider the relaxations of path and twig queries specific to personal information search in file systems. In particular, we introduce specific relaxations (e.g., node permutations) for the queries in a directory-based file system.

2.5 Indexing Methods and Query Processing

Various XML indexing methods are summarized and compared in [62, 83]. Our structure indices are an extension of the inverted list from information retrieval community. They store a mapping from the labels to their locations in a data tree that combines the directory structure and internal file structure. More specifically, if only directory structures are used for search, we encode the positional information in the data tree using directory IDs and depths. If both the directory structures and internal file structures are needed, we use the *PrePost* coding to represent positions in the data tree.

Several works such as [18, 64] have studied efficient path matching techniques for XML documents. In [64], path queries are decomposed into simple subexpressions and the results of evaluating these subexpressions are combined or joined to get the final result. In [18], a *PathStack* is constructed to track partial and total answers to a query path by iterating through document nodes in their position order. The works on partial path queries [87] extend *PathStack* and allow fuzziness in the query conditions by keeping some structural relationships between query nodes undefined. While our work is also based on *PathStack*, we support more complex path queries containing term permutations using dedicated query structures.

The authors in [51] classify the existing XML query processing methods into three categories: the file approach [3, 17, 18, 35–37, 55], the relational approach [23, 46, 68, 75, 76, 90, 92], and the native approach [2, 18, 57, 91, 92]. In this thesis, we primarily focus on the path queries. The query processing for twig queries in future work may use some ideas from the XML query processing methods.

The Threshold Algorithm (TA) [43] is a popular method to combine values from multiple sorted lists and evaluate top-k answers. We use *TA* to aggregate scores for multiple query conditions. In our scenario, *TA* uses a threshold condition to avoid evaluating all possible matches to a query, focusing instead on identifying the k best answers.

2.6 XML Scoring

Earlier research [7, 16, 25, 45, 49, 61, 71, 74, 77, 81, 83, 85] does not consider structural relaxations while scoring both structure and content conditions. We use ideas introduced in [6] that considers relaxations; however, there are several important differences. Specifically, in [6]the matches of XML queries are defined for the root node of twig queries while ours are defined for the leaf nodes of path or twig queries. As a result, the scores are different since the matches are not the same. Further, as discussed in later chapters, our unified structure and content scoring is defined across all relaxed queries. The lexicographical ordering of (*idf, tf*) proposed in [6] is an approximation to our unified rank ordering.

The INitiative for the Evaluation of XML retrieval (INEX) [56] promotes new scoring methods and retrieval techniques for XML data. INEX provides a collection of documents as a testbed for various scoring methods in the same spirit as TREC was designed for keyword queries. While many methods have been proposed in INEX, they focus on content retrieval and typically use XML structure as a filtering condition. As a result, the INEX datasets and queries would need to be extended to account for structural heterogeneity. Therefore, they could not be used to validate our scoring methods. We developed our own ranking benchmarks based on the real data from our personal file systems.

As part of the INEX effort, XIRQL [49] presents a content-based XML retrieval query language based on a probabilistic approach. While XIRQL allows for some structural vagueness, it only considers edge generalization, as well as some semantic generalizations of the XML elements. Similarly, JuruXML [20] provides a simple approximate structure matching by allowing users to specify path expressions along with query keywords and modifies vector space scoring by incorporating a similarity measure based on the difference in length, referred to as length normalization.

Integrating content and structure score is a complex task that requires a better understanding of the interconnections between structure and content. Efforts in the XML community have been made in this direction [20,49] on a simpler set of structural relaxation rules; our techniques use new relaxation rules to achieve finer grained control to searching personal file system.

Chapter 3

Multi-dimensional Search

In Chapter 1, we pointed out numerous search tools have been developed as a combination of keyword search and filtering for non-keyword conditions (e.g., file directory, date, file type). Unfortunately, simple keyword-only searches plus filtering are often insufficient.

In Example 1, John could ask the query:

```
[createdDate=11/01/2008 AND
 content="Halloween witch" AND
 filetype=*.jpg AND
 structure=/Desktop/Pictures/home]
```

Current tools would answer this query by returning all files of type **.jpg* created on 11/01/2008 under the directory */Desktop/Pictures/home* (filtering conditions) that have content similar to “Halloween witch” (ranking expression), ranked based on how close the content matches the text “Halloween witch” using some underlying text scoring mechanism. Because the date, directory structure, and file type are used only as filtering conditions, files that are very relevant to the content search part of the query, but which do not satisfy these exact conditions would not be considered as valid answers. For example, **.tif* files created on 10/31/2007, or relevant files in the directory */archive/Pictures/home* would not be returned.

We believe that allowing flexible conditions on structure and metadata can significantly increase the quality and usefulness of search results in many search scenarios. For instance, for the above example query, the user might not remember the exact creation date of the file of interest but remembers that it was created *around* 11/01/2008. Similarly, the user might be primarily interested in files of type **.jpg* but might also want to consider relevant files of different but related types (e.g. **.tif* or **.gif*). Finally, the user might misremember the directory path under which the file was stored. In this case, the date, file type, and direcotry conditions should not only be considered for filtering purposes but should also be part of the ranking conditions of the queries.

The challenge is then to adequately score the search results by taking into account flexibility in the textual component *together* with some flexibility in the structural and metadata components of the query.

In this chapter, we present a multi-dimensional approach that allows users to provide fuzzy conditions on three query dimensions: content, metadata, and structure. We describe individual *IDF*-based scoring approaches for each dimension and present a unified scoring framework for multi-dimensional queries over personal information file systems. Our techniques are based on an *IDF*-based interpretation of scores for each dimension. There has been some discussions in both the Database and the Information Retrieval communities on integrating technologies from both fields [1,4,9,21] to combine content-only searches with structure-based query results. Our techniques provide a step in this direction as they integrated IR-style content scores with DB-style structure approximation scores.

While very relevant to this section, the work on scoring strategies for the metadata information was done outside the context of this thesis. We will review the approach presented in this work in so far as it is relevant to presenting a unified scoring approach.

Further, while the work presented in this chapter could be extended to a variety of dataspace applications and queries, we focus on the file search scenarios. That is, we consider the granularity of the search results to be a single file in the personal information system. Of course, our techniques could be extended to a more relaxed query model where subset of files (such as individual sections or XML subtrees) could be returned as a result.

In this chapter, we focus on scoring strategies. The details of our implementation of query optimization strategies will be discussed in Chapter 4.

For the rest of the chapter, we first present the details of our multi-dimensional scoring framework in Section 3.1. We then present the detailed experimental results in

Section 3.2.

3.1 A Unified Multi-dimensional Scoring Framework

In this section, we present our unified framework for assigning scores to files based on how closely they match individual query dimension conditions. We distinguish three scoring dimensions: *content* for conditions on the textual content of the files, *metadata* for conditions on the system information related to the files, and *structure* for conditions on the directory path to access the file.

Our scoring strategy is based on an *IDF*-based interpretation of scores for each dimension. Traditionally, the *IDF* score of a document for a keyword in the IR world is a function of how many documents contain the keyword [84]. The content *IDF* scoring strategy has been widely adopted in IR systems as it considers the distribution of answers to assign scores. We extend this idea to each of our search dimension and assign a score to a file in a query dimension based on how many files match the query condition. The unification aspect of our scoring framework comes from this *IDF*-based scoring approach; for each dimension, the score of a file is a function of the document frequency. The multi-dimensional score of the file is then a combination of the individual dimension scores. It is our belief that using a unified *IDF* framework allows us to meaningfully combine scores on several orthogonal dimensions to provide a single result, as we will show experimentally in Section 3.2.

We first give a brief overview of our query model (Section 3.1.1), we then present our *IDF*-based scoring strategies for each dimensions: content (Section 3.1.2), metadata (Section 3.1.3), and structure (Section 3.1.4). Finally, we show how we aggregate scores across dimension in Section 3.1.5.

3.1.1 Query Model

To perform multi-dimensional queries, we need a query language that can express metadata and structure conditions in addition to content searches. To this end, we use the simplified version of XQuery [88] as our query language. The path structure in each query condition indicates the type of data (i.e., content, metadata, or structure) being queried. For example, query conditions containing the component "Structure" in their paths indicate queries over structure information. The query from our earlier example would be expressed as follows:

```
FOR $i in /File[FileSysMetadata/FileDate = '11/01/08']
  FOR $j IN /File[ContentSummary/WordInfo/Term = 'Halloween'
                  AND ContentSummary/WordInfo/Term = 'witch']
    FOR $m IN /File[FileSysMetadata/FileType = 'jpg']
      FOR $n IN /File[Structure = '/Desktop/Pictures/home']
      WHERE $i/@fileID = $j/@fileID AND
            $i/@fileID = $m/@fileID AND
            $i/@fileID = $n/@fileID
      RETURN $i/fileName
```

An answer to a query is a file that is relevant to one or more of the query conditions. Internal flags are used to specify whether only exact matches are allowed (filtering conditions) or whether approximate matches are considered (ranking conditions). If approximate matches are allowed, a score is assigned for each query condition based on how closely the answer (file) matches the condition (exact matches for filtering conditions have a score of 1).

In the rest of this section, we discuss our strategies for scoring approximate matches for query conditions in each dimension.

3.1.2 Scoring Content

We use standard indexing structures and scoring mechanisms from the IR literature [84] for query conditions involving text content. Specifically, we assign scores using a standard $TF \cdot IDF$ scoring function. Relaxation is already an integral part of $TF \cdot IDF$ since it scores files that contain only a subset of the terms as well as those containing all terms in the content query condition.

Definition 1 (Content TF·IDF Score of a File) *For a given keyword query, Q,*

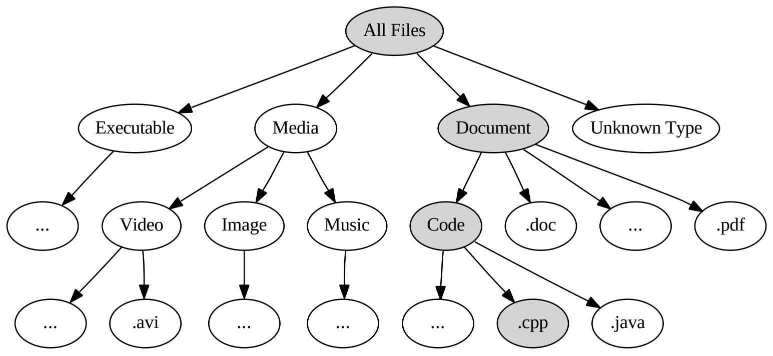


Figure 3.1: *Fragments of the indexing DAGs for file type (extension) metadata. It represents portions of the complete DAGs with several levels removed for simplicity of presentation. Highlighted nodes indicate the sequence of relaxations for a file type query of ".cpp".*

consisting of the terms $t_1,\ t_2,\ \ldots, t_n$, the content score of a file F, with respect to Q is computed as:

$$score_{Content}(Q,F) = \frac{\sum_{i=1}^{n}(IDF_{t_i} \cdot TF_{t_i,F})}{\sqrt{|F|}}$$

with

$$TF_{t,F} = 1 + \log(F_t) \qquad\qquad IDF_t = \log(1 + \frac{N}{N_t})$$

where $|F|$ is the total number of terms in the file, F_t is the number of times the term t appears in file F, N_t is the number of files containing the term t, and N is the total number of files.

Note that to stay consistent with traditional IR system, in this dimension we consider the *TF* score of a match as well as its *IDF* score. This stays in the spirit of our overall *IDF*-based framework, as the *TF* score is only used to give additional weight information on the quality of the matches.

3.1.3 Scoring Metadata

Dataspaces and personal information systems allow for the storage of metadata information alongside files. Such metadata may include file sizes, file owners, and various file timestamps (e.g., date created and date last modified). File extensions can also hint at the corresponding file types. Users often want to enhance their query with metadata conditions (e.g., file was accessed last week, file is a pdf document), but may not accurately remember the exact metadata values for which they are looking. Therefore, allowing for some approximation in metadata conditions is desirable.

We introduce a hierarchical relaxation approach for each type of searchable metadata to support scoring. For example, Figure 3.1 shows (a portion of) the relaxation levels for file types, represented as a DAG. Each leaf represents a specific file type (e.g., pdf files). Each internal node represents a more general file type that is the union of the types of its children (e.g., *Media* is the union of *Video*, *Image*, and *Music*) and thus is a relaxation of its descendants. A key characteristic of this hierarchical representation is *containment*; that is, the set of files matching a node must be equal to or subsume the set of files matching each of its children nodes. This ensures that the score of a more relaxed form of a query condition is always less than or equal to the score of a less relaxed form (see Equation 3.1 below).

We then say that a metadata condition *matches* a DAG node if the node's range of metadata values is equal to or subsumes the query condition. For example, a file type query condition specifying a file of type "*.cpp" would match the nodes representing the files of type "Code", the files of type "Document", etc. A query condition on the creation date of a file would match different levels of time granularity, e.g., day, week or month. The nodes on the path from the deepest (most restrictive) matching node to the root of the DAG then represent all of the relaxations that we can score for that query condition. Similarly, each file *matches* all nodes in the DAG that is equal to or subsumes the file's metadata value.

Finally, given a query Q containing a single metadata condition M, the metadata score of a file f with respect to Q is computed as:

$$score_{MetaData}(Q,f) = \frac{log(\frac{N}{nFiles(commonAnc(n_M,n_f))})}{log(N)} \tag{3.1}$$

where N is the total number of files, n_M is the deepest node that matches M, n_f is the deepest node that matches f, $commonAnc(x,y)$ returns the closest common ancestor of nodes x and y in the relaxation hierarchy, and $nFiles(x)$ returns the number of files that match the relaxation level of node x in the file system. The score is normalized by $log(N)$ so that a single perfect match would have the highest possible score of 1.

3.1.4 Scoring Structure

Most users organize their files into a hierarchical directory structure for navigation. In addition, the structure *within* a document can be seen as an extension of the directory path structure and used for more complex query searches [38]. However, users are notoriously bad at remembering where they stored a particular file or how the files are structured [30]. When a user searches for a file using structure information such as directory path information, the query is likely to be incorrect, as users often confuse or misremember the order of the directories, their relationships, or their labels. However, it is common that users do correctly remember some portion of the path whether it be a prefix or several (possibly non-consecutive and out-of-order) directory names. Therefore, allowing for a method of approximation that leverages any correct information in an otherwise incorrect (when taken as a whole) path is desirable.

In this section we discuss our scoring strategies for the structure information of files. While our main focus in this chapter is on directory path structure, we will adapt our structure scoring techniques for structural information within a document in Chapter 5. Our structure scoring strategy is based on work on XML structural query relaxations [5–7]; as described below, we introduce new types of structural relaxations to handle the specific needs of user searches in a personal information management system.

3.1.4.1 Notations

We first introduce a few notations that we use to define relaxations of directory structure query conditions.

- **Directory Tree:** A directory tree D is a hierarchical representation of a navigational directory structure, rooted at the top of the directory hierarchy, and where each directory is a node in the tree.

- **Path Node:** Given a directory tree D, a path node N is either: (a) a single directory with a given label l, (b) the root of the directory tree, or (c) a wildcard directory (i.e., a directory with any possible label) noted $*$.

- **Path Query:** A path query P is a simple non-cyclic path where each node in the path is either a path node or a node group (see below) and each edge in P is either a parent-child edge (/) or an ancestor-descendant edge (//).

  For example, $root/a//b$, $root//a/b//(c/d)$, and $root//a/b//*$ are path queries. For simplicity, we consider $/a//b$, $//a/b//(c/d)$, and $//a/b//*$ to be equivalent to $root/a//b$, $root//a/b//(c/d)$ and $root//a/b//*$ respectively.

- **Node Group:** To represent possible permutations in path queries, we introduce the notion of node groups. A node group is a simple non-cyclic path where all nodes are (labeled) nodes and each edge in the path is either a parent-child edge or an ancestor-descendant edge. The placement of edges is fixed within the group, however the (labeled) nodes may permute. The *extension* of a node group n is the set of all path queries that are contained in n and do not contain any node groups. Note that by definition the root node is always at the head of a path query and a wildcard node is always at the tail of a path query. Thus, the *root* node and * node are not allowed in a node group. Any other definitions or relaxation rules should always keep this property.

  For example (a/b) is a node group, which corresponds to the extension set containing a/b and b/a, and $(a//b/c)$ is a node group, which corresponds to the extension set containing $a//b/c$, $a//c/b$, $b//a/c$, $b//c/a$, $c//a/b$ and $c//b/a$.

The set of exact answers to a path query P is the set of files that can be reached through the path(s) defined in P, i.e., P and any extensions of P. The set of all answers to P is the set of files that can be reached through path(s) defined in P or in a relaxation of P as defined below.

3.1.4.2 Structure Relaxations

We consider four different relaxation operations to derive approximations of structural query conditions. While our relaxations are inspired from the work on XML structural relaxation [5–7], our needs differ from work on approximate XML queries in two significant ways:

- We consider path queries instead of twig queries. As such, we do not need specific twig relaxations such as subtree promotion [5]. In contrast, we need to be able to delete or extend any directory node contained within the path query to include any files included in the directory subtree rooted at the directory node.
- Permutations of nodes within a path query is very common in file search scenarios. For this purpose, we introduce the node inversion operation, as well as the node group notation. Most work on XML query relaxation have ignored node inversion, although they may be able to simulate some node inversion cases with relaxed twig query patterns [6, 7].

As in [6, 7], we require that answers to a path query P be contained in the set of answers to a relaxation of P to ensure monotonicity of scores when relaxing a query (since scores depend on the number of files that are answers to the path query). We consider the following four structural relaxation operations:

- **Edge Generalization** is used to relax a parent-child edge to an ancestor-descendant edge. It can be used in a path query or a node group.
- **Path Extension** is used to extend a path query P that does not end in $//*$ to $P//*$ so that all files within the directory subtree rooted at P can be considered as answers. Answers to paths that do not end in $//*$ only return files that are located in the exact directory rooted at P (and at the extensions of P).
- **Node Deletion** is used to drop a node from a path query. Node deletion can be applied to any path query or node group but cannot be used to delete the *root* node or the $*$ node.

- To delete a node n in a path query P:
  - If n is a leaf node, n is dropped from P and $P - n$ is extended with $//*$. This is to ensure containment of the answers to P in the set of answers to P', and monotonicity of scores.
  - If n is an internal node, n is dropped from P and $parent(n)$ and $child(n)$[1] are connected in P with $//$.

  For example, deleting node c from $a/b/c$ results in $a/b//*$ because $a/b//*$ is the most specific relaxed path query containing $a/b/c$ that does not contain c. Similarly, deleting c from $a/c/b//*$ results in $a//b//*$.

- To delete a node n that is within a node group N in a path query P, the following steps are required to ensure answer containment and monotonicity of scores:
  - n and one of its adjacent edge in N are dropped from N. Every edge within N becomes an ancestor-descendant edge. If n is the only node left in N, N is replaced by that node in P.
  - Within P the surrounding edges of N are replaced by ancestor-descendant edges.
  - If N is a leaf node group, the result query is extended with $//*$.

  For example, deleting node a in $x/(a/b//c/d)/y$ results in $x//(b//c//d)//y$ because the extension set of $x/(a/b//c/d)/y$ contains 24 path queries, which include $x/a/b//c/d/y$ and $x/b/c//d/a/y$; after deleting node a, these two path queries become $x//b//c/d/y$ and $x/b/c//d//y$. Therefore, $x//(b//c//d)//y$ is the only most specific path query which contains the complete extension set and does not contain a.

- **Node Inversion** is used to permute nodes within a path query P. Permutations can be applied to any adjacent nodes or node groups except for the *root* and $*$

[1]For simplicity, $parent(n)$ and $child(n)$ denote the nodes at the parent and child positions of n respectively. The edges connecting $parent(n)$ (or $child(n)$) and n could be either a parent-child or ancestor-descendant edge.

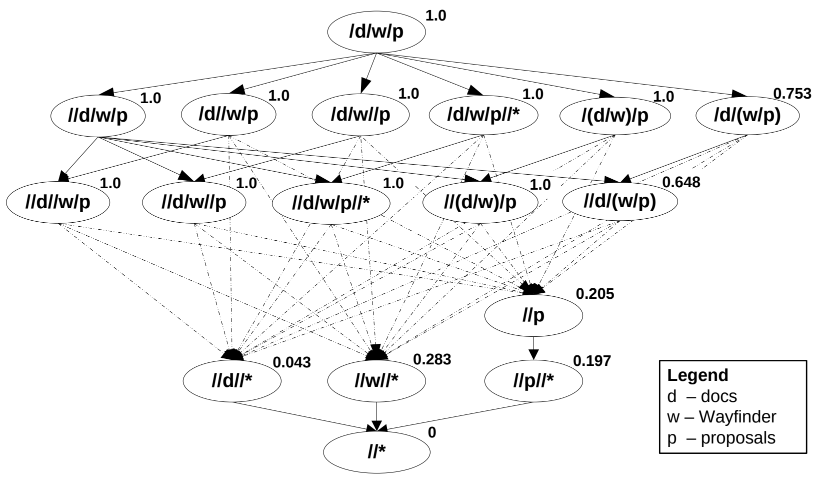


Figure 3.2: The structure relaxation DAG for an example structural query condition */docs/Wayfinder/proposals*. Solid lines represent parent-child relationships. Dotted lines represent ancestor-descendant relationships, with intermediate nodes removed for simplicity of presentation. IDF scores are shown at the top right corner of each DAG node.

nodes. A permutation combines adjacent nodes, or node groups, into a single node group while preserving the relative order of edges in P. For example, applying node inversion to nodes a and b in $x/a/b/y//*$ results in $x/(a/b)/y//*$.

Our relaxation operations can be composed to provide increasingly relaxed versions of the original path query. For any path query P, the most general relaxation is $//*$ and $//*$ matches any path in the directory tree.

Note that path extensions and node inversions were not considered in [6,7]. In particular, node inversions significantly complicate and increase the number of possible query relaxations and require the introduction of the node group notation (Section 3.1.4.1) to represent path permutations.

3.1.4.3 DAG Representation of Structure Relaxations

As proposed in [6], we use a DAG to represent all possible structural relaxation of a path query condition. The DAG structure is used not only to compute and store score information but also for query processing, as it allows us to incrementally access increasingly relaxed answers during query processing. Figure 3.2 shows an example relaxation DAG, along with example *IDF* scores (see Section 3.1.4.4 for details), for the structure query condition */docs/Wayfinder/proposals*. This DAG is rooted at the exact query condition itself, with each non-root node representing a relaxed form of the exact query condition. Note that this *structure* DAG is *query specific* in that it is a representation of the query and all possible relaxed forms of that query given the above relaxation operations, rather than an index of data in the file system like the indexing metadata DAGs in Figure 3.1. Matches for the exact query $/d/w/p$ have a score of 1, while matches to increasingly relaxed versions of the query, as we go down the DAG, have lower scores, with matches to the most general relaxation of $/d/w/p$: $//*$ having a score of 0.

We present Algorithm 1 to build the DAG in a top-down fashion given a path query P. The DAG creation algorithm starts by creating a node containing the exact path query P and incrementally applies simple relaxation steps to create new DAG nodes, merging identical DAG nodes on the fly.

Algorithm 1 *buildPQDAG*(*currentDAGNode*)

```
1.  $P = getQuery(currentDAGNode)$
2.  **for** each edge $e$ in $P$ **do**
3.    **if** $isParentChildEdge(e)$ **then**
4.      $newDAGNode = getDAGNode(edgeGeneralize(e, P))$
5.      {getDAGNode checks if a DAG node containing $P$ with the edge generalization exists,
        and creates such a DAG node if it does not.}
6.  **for** each node or node group $n$ in $P$ **do**
7.    **if** not $isWildcardNode(n)$ **and** $parent(n)$ exists **and**
           not $isRootNode(parent(n))$ **then**
8.      $newDAGNode =$
9.          $getDAGNode(invertNode(n, parent(n), P))$
10. **for** each node or node group $n$ in P **do**
11.   **if** $requireNodeDeletion(n)$ **then**
12.     **if** $isNode(n)$ **then**
13.       $newDAGNode = getDAGNode(nodeDeletion(n, P))$
14.     **else**
15.       {$n$ is a node group.}
16.       **for** each node $m$ in $n$ **do**
17.         $newDAGNode =$
18.             $getDAGNode(nodeDeletion(m, P))$
```

To ensure the relaxation is done incrementally, we create an edge from a DAG node P to a more relaxed DAG node P' if and only if there is no DAG node P'' such that P'' is a relaxed query of P and P' is a relaxed query of P''. To achieve this, Algorithm 1 always applies edge generalization, path extension, and node inversion first when possible. Node deletion is only applied when no other relaxation is possible anymore.

The function $requireNodeDeletion(n)$ checks whether a node deletion relaxation needs to be applied. Node deletion is only applied on a path node n if its surrounding edges are ancestor-descendant edges. If n is a node contained in a node group, node deletion is only applied if all surrounding and internal edges of the node group are ancestor-descendant edges.

3.1.4.4 Scoring Structural Relaxations

To score files based on approximate structural conditions, we use an adaptation of *IDF* to structure relation that is similar to the one presented in [6].

Definition 2 (Structural IDF of a Path Query) *For a path query P and a directory tree D, the IDF score of any file f in D that is a valid answer to P is computed as:*

$$IDF_D(P, f) = \frac{log(\frac{N}{|F(D,P)|})}{log(N)}$$

where N is the total number of files in D and $F(D, P)$ is the set of all files that match P and its extensions in D.

Figure 3.2 shows an instance of the normalized *IDF* scores corresponding to each DAG node path query relaxation for an example data instance. All files that match the same specific path relaxations have the same structure *IDF* score. Note that in Figure 3.2 path queries $/d/w/p$, $//d/w/p$, and $//d//w/p$ have exactly same *IDF* scores. This occurs because the number of files that can be reached through them and their extensions in D are the same (Definition 2). Because answers to a path query P are contained in the set of answers to any relaxation of P, this implies that $/d/w/p$, $//d/w/p$, and $//d//w/p$ have the same set of answers, i.e., these relaxations do not result in more approximate matches.

Definition 3 (Structure Score of a File) *For a given structure query Q, the structure score of a file f, with respect to Q is computed as:*

$$score_{Structure}(Q, f) = \max_{P' \in R(Q)} \{IDF_D(P', f) | f \in F(D, P')\}$$

where $R(Q)$ is the set of all possible relaxations of Q, $F(D, P')$ is the set of all files that match a relaxation P' of Q or the extensions of P' in D.

Intuitively, the *IDF* of a file $f \in D$ that is an (possibly approximate) answer to Q is the number of all possible paths in D divided by the number of answers for the most specific relaxation of Q for which f is a valid answer.

3.1.5 Aggregating Multi-dimensional Scores

A strength of our scoring framework is that all dimensions are scored using a similar *IDF* metric, which takes into account the number of files that match a particular query condition (or relaxation of that condition). This unified framework allows us to meaningfully aggregate scores across different query dimensions.

We aggregate the individual dimension scores to produce the final score of a file for a query. A vector projection is used for the aggregation of multi-dimension scores. We build a 3-dimension file vector $\vec{V_F}$, which consists of the (normalized) three dimension (content, metadata, and structure) scores. For a query Q and a file F, we have:

$$\begin{aligned} \vec{V_F} &= (score_{Content}(Q, F), score_{MetaData}(Q, F), \\ &\quad score_{Structure}(Q, F)) \end{aligned}$$

The query vector $\vec{V_Q}$ has value 1 (exact match) for each dimension. The file multi-dimensional score is the normalized length of the projection of the document vector on the query vector.

Definition 4 (Query Score of a File) *For a given query, Q with corresponding query vector $\vec{V_Q}$, the score of a file F with corresponding document vector $\vec{V_F}$, with respect to Q is computed as:*

$$score(Q, F) = \frac{\vec{V_F} \cdot \vec{V_Q}}{|\vec{V_Q}|}$$

Note that our aggregation assigns the same importance to each dimension in the query. We could easily incorporate weights in our aggregation function to give more importance to one or more dimension.

3.2 Experimental Evaluation

In this section, we will experimentally validate our *IDF*-based scoring approach and evaluate the potential for the corresponding multi-dimensional fuzzy search approach to improve relevance ranking. We also report on query performance to show that multi-dimensional search is practical to use.

3.2.1 Experimental Setting

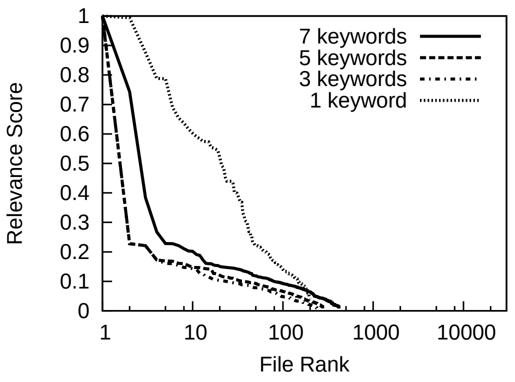
3.2.1.1 Platform

All experiments were performed using the Wayfinder file system [69]. Experiments were run on a PC with a 64-bit hyper-threaded 2.8 GHz Intel Xeon processor, 2 GB of memory, and a 10K RPM 70 GB SCSI disk, running the Linux 2.6.16 kernel and Sun's Java 1.5.0 JVM.

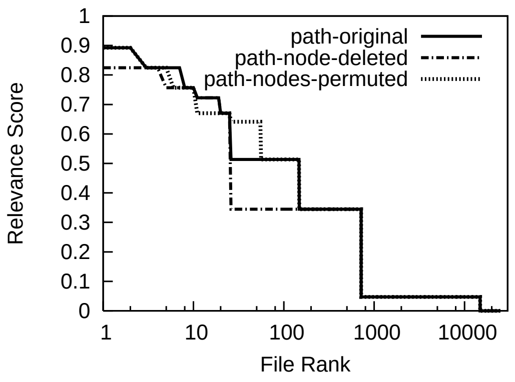
3.2.1.2 Data Set

As noted in [38], there is a lack of synthetic data sets and benchmarks to evaluate search over personal information management systems; therefore we used a real user data set comprised of (a representative subset of) files and directories from the working environment of a user in our lab. This data set contained 24,927 files in 2,339 directories. 24% of this data set were multi-media files (e.g., music and pictures), 17% document files (e.g., pdf, text, and MS Office), 14% email messages,[2] and 12% source code files. The average directory depth was 3.4 with the longest being 9. On average, directories

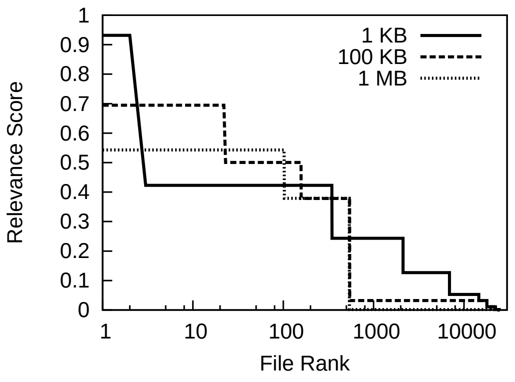
[2]Email messages are stored in the Maildir format in which each email is stored in a separate file.



(a) Content Queries



(b) Structure Queries



(c) Size Metadata Queries

Figure 3.3: *Relevance score plotted as a function of ranking for (a) four content queries, (b) three structure queries, and (c) three size metadata queries.*

contained 11.6 sub-directories and files, with the largest—a folder containing emails—containing 1013. Wayfinder extracted 347,448 unique stemmed content terms.[3] File modification dates spanned 10 years. 75% of the files were smaller than 177 KB, and 95% of the files were smaller than 4.1 MB.

3.2.2 Behaviors of Scoring Functions

We start by studying the behaviors of our scoring functions. Figure 3.3 plots the scores of all relevant files as a function of their ranking for several 1-dimensional queries targeting three different dimensions as follows. Figure 3.3(a) plots the normalized scores for four content queries that contain from one to seven keywords. For each query, the scores of matching files were normalized against the highest score since content scoring

[3]Content was extracted from MP3 music files using their ID3 tags.

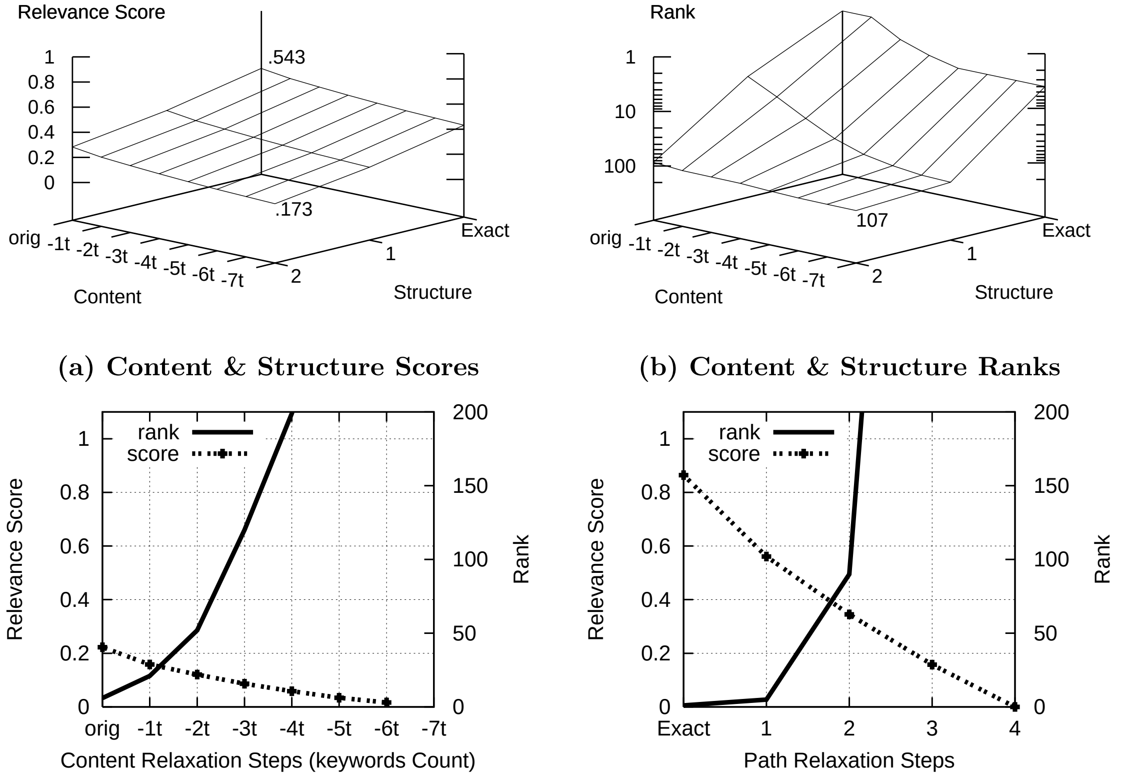
is based on $TF \cdot IDF$ as opposed to just *IDF*. Figure 3.3(b) plots the scores for three structure queries. The first query is a standard path query that corresponds to a path that exists in the directory tree (of the form */a/b/c/d*), the second is the original path query after deleting a node (*/a/b/d*), and the third is the original path query after permuting two nodes (*/a/b/d/c*). We choose these queries to exhibit common user mistakes in querying structure. Finally, Figure 3.3(c) plots the scores for three metadata queries for three different file sizes.

First and foremost, we observe that all scoring functions have similar behaviors. In all cases, the relevance scores monotonically decrease (by design) as files become less similar to the query conditions. Most critically, all scoring functions allow a large number of files that do not exactly match the query conditions to be scored and ranked, providing the desired flexibility over filtering. This is particularly important when a query condition such as a *non-existing* directory is provided in the query; in such cases, filtering would not consider any file in the system as being relevant to the query.

On the other hand, there are several interesting differences between the scoring function for content and the scoring functions for the other dimensions. In particular, the scoring functions for structure and metadata (Figures 3.3(b-c)) are noticeably plateau-shaped because of our DAG-based approach to computing IDF. In this approach, each relaxation step is likely to bring a set of files that are deemed to be equally similar to the query condition. For instance, in Figure 3.3(c), each plateau corresponds to a discrete relaxation interval to which a range of file sizes has been mapped.

Plateaus in the scoring function, where many files are assigned the same score, can make a query dimension less useful for ranking. For metadata, we can arbitrarily smooth the scoring function as long as files do not have exactly the same attributes (e.g., same size) by considering increasingly smaller relaxation intervals. In contrast, this is not possible to do for structure, as the relaxation intervals are defined by the matching directories and the number of files inside each directory, which is under user control. (Users can aid the search engine by using sparser directory structures.)

The content scoring function also differ from the other two in its sharp drop from the top several ranked results to the 10th-100th ranked results and its non-zero scoring



(a) Content & Structure Scores

(b) Content & Structure Ranks

(c) Content

(d) Structure

Figure 3.4: *Score (a) and rank (b) of a target file returned as a result of a constant 2-dimensional query as the file is relaxed away from the query conditions across the two query dimensions. Score and rank of the same file target file plotted for a content-only query (c) and a structure-only query (d).*

of a much smaller subset of the files in the system. These differences do not seem fundamental, however. For example, if we choose a set of content terms that appear in most files, then the content scoring function would likely look more like the other two.

Thus, despite the differences, we conclude that the data in Figure 3.3 makes a strong case for combining relevance scores from orthogonal query dimensions using our framework as it places these dimensions into a common setting to be compared. The differences mentioned likely provide opportunities to explore more complex score aggregation approaches in future work.

3.2.3 Scores and Rankings for Approximate Answers

We now show how scoring (and the corresponding ranking) is affected by inaccurate query conditions. For this purpose, we choose a target file that is an exact match for the query conditions of a particular query, resulting in a high score and rank. We then modify the target file so that it is progressively farther away from the query conditions; that is, the file will only match increasingly relaxed approximations of the query conditions. While relaxing the target file, we alter several other files as needed to ensure that any global statistics used in scoring computations are kept constant. This ensures that scores for files unrelated to our relaxation process are unaffected, providing a stable background for interpreting the changing score (and rank) of the target file.

Figures 3.4(a-b) plot the score and rank, respectively, of the target file for a representative 2-dimensional queries covering the content and structure dimensions. In the content dimension, we relax the file by progressively removing occurrences of the query term from within the file. In the structure dimension, we relax the file by progressively moving the file up the directory tree (representing simple relaxations steps) from its original location.

It should be noted that in Figure 3.4, the target file is not the only exact match to the query condition in the structure dimension, leading to a relevance score of less than 1 even before relaxation of the file. Also, the target file is not returned as the top result to the content-only query leaving again a score of less than 1. This arises from the fact that our data set has several small files that contain a subset of the query terms. As our $TF \cdot IDF$ scores are normalized by file lengths, these smaller files achieve higher scores than the target file, which is a novel containing over 100,000 terms.

Figures 3.4(a) show that the combined scoring functions for a multi-dimensional query behave as expected; that is, they preserve the trends of the 1-dimensional scoring functions, decreasing as the file is relaxed away from the query conditions in either dimension; we plot the score and rank of the target file as it is relaxed against 1-dimensional content and structure queries in Figures 3.4(c-d) for comparison purposes. Although the results are not shown here, scoring for 2-dimensional metadata queries

Query Evaluation Results						
	Query Conditions					**Comments on Relaxation**
Query	**Content**	**Type**	**Date**	**Structure**	**Rank**	**from Query Q1**
Q1	C	-	-	-	49	Base Query
Q2	C	.txt	26 Feb 07 16:08	$/p/e/n/j$	1	Correct Values (all dim.)
Q3	C	.txt	-	-	6	Correct Value
Q4 *	C	.pdf	-	-	1026	Incorrect Value
Q5 *	C	.doc	-	-	45	Incorrect Value
Q6	C	Docs.	-	-	21	Relaxed Range
Q7	C		26 Feb 07	-	5	Relaxed Range (Day)
Q8	C	-	25-28 Feb 07	-	5	Relaxed Range (Week of month)
Q9	C	-	Feb 07	-	7	Relaxed Range (Month)
Q10 *	C	-	27 Feb 07 16:08	-	9	Incorrect Value (off by 1 day)
Q11 *	C	-	19 Feb 07 16:08	-	14	Incorrect Value (off by 1 week)
Q12 *	C	-	26 Mar 07 16:08	-	150	Incorrect Value (off by 1 month)
Q13	C	-	-	$/p/e/n/j$	3	Correct Path
Q14	C	-	-	$/p/e$	13	Prefix of Correct Path
Q15 *	C	-	-	$/j/e$	3	Incorrect Order/Correct Names
Q16 *	C	-	-	$/p/e/n/h$	11	Incorrect Path
Q17	C	Docs.	Feb 07	$/p/e/n$	3	Relaxed Range (all Dim.)
Q18 *	C	.pdf	19 Feb 07 16:08	-	36	Incorrect Values
Q19 *	C	.pdf	19 Feb 07 16:08	$/j/e$	2	Incorrect Values (all Dim.)

Table 3.1: *The rank of a target file—the novel Sea Wolf by Jack London—returned by a set of related queries. In the presences of ties in the relevance scores, the highest rank the target file could have is given. The queried dimensions include* Content, Type *(Metadata),* Date *(Metadata), and* Structural. *The initial content query Q1 provides the set C containing the 4 query terms* {jack, london, sea, wolf}. *Structural values are abbreviated. The complete path of our target file is* /Personal/Ebooks/Novels/JackLondon/. *Queries which contain a "*" in the first column represent those in which the target file would not be considered as a relevant answer given today's typical filtering approach.*

and 3-dimensional queries also display similar behaviors.

More interestingly, we observe that providing query conditions for other dimensions in addition to content, even when the provided query values are somewhat inaccurate, can significantly improve ranking accuracy. For example, when the target file contains only 5 of the 7 terms (-2t) in the content query, its rank drops to around 50 (Figure 3.4(c)). When we provide an approximate structural value, the parent directory of the directory containing the target file, the ranking jumps to close to 10 (Figure 3.4(b)).

Of course, providing incorrect query values can hurt ranking as well. For example, providing an ancestor directory two level up pulls the rank of the target file down to around 100, even when the file contains all 7 query terms in the content dimension. Keep in mind, however, that providing incorrect non-content query values to current filtering approaches may prevent the target file from being ranked at all. Thus, in these cases, our approach may not improve on the current filtering approaches (users typically do not look at returned results beyond some top K ranked items) but is no worse than filtering.

3.2.4 Impact of Flexible Multi-Dimensional Search

In the last section, we shown the general trends of multi-dimensional scoring to validate that our combined scoring function behaves as desired. We also argued that providing fuzzy query conditions in non-content dimensions has the potential to significantly improve scoring (and thus ranking) accuracy. In this section, we explore this latter potential and compare our approach against current filtering approaches in more detail.

In this study, we initially construct a content-only query intended to retrieve a specific target file and then expand this query along several other dimensions. We start with a base content-only query because content-only queries are the standard search interface in many real world systems. For each query, we consider the ranking of the target file by our approach together with whether the target file would be ranked at all by today's typical filtering approaches on non-content query conditions.

Table 3.1 summarizes the results of our study. The target file is the novel Sea Wolf by Jack London and the set of query content terms, C, in our initial content-only query, Q1, contains the four terms *sea*, *wolf*, *jack*, and *london*. While the query is quite reasonable, the terms are generic enough that they appear in many files, leading to a ranking of 49 for the target file. Query Q2 augments Q1 with the exact matching values for file type, date, and containing directory. This brings the rank of the target file to 1. Of course, this result by itself is not meaningful because it is unlikely that the user will remember the file attributes with such precision.

In the remainder of the table, we explore what happens when we modify the query in two different ways for the non-content dimensions: (1) instead of the precise correct value we provide a range around the precise value, and (2) we provide an incorrect value.[4] The results are quite promising. For example, in query Q15, just getting

[4]We do not consider incorrect ranges, that is, a range that does not include the value of the target file, because the results are similar to incorrect values; filtering would not rank the target file and in our approach, while the scores change, the ranking does not change significantly.

a couple of components correct in the directory name—note that the components are given in an incorrect order—brings the ranking up to 3. Providing an incorrect directory that shares a common prefix with the correct directory brings the ranking to 11 (query Q16). In contrast, if such directories were given as filtering conditions, the target file would be considered irrelevant to the query and not ranked at all.

Similar results can be seen for most other queries marked with a "*" in the first column (indicating that the target file would not be found using a filtering-only approach). Two exceptions include queries Q4 and Q12. In these queries, the incorrect values for the other dimensions reduce the ranking of the target file below that achievable with only content. For query Q4, this decreased ranking is because there are many pdf documents that achieve a higher metadata score than the target file. Similarly, for query Q12, many files with dates closer to the query condition achieve higher metadata scores. Given that the ranking in these two cases are 1026 and 150, our approach is not meaningfully different from filtering since users are unlikely to look that far down a ranking list.

Using ranges also give promising results although our approach is unlikely to outperform filtering (when the matching value of the file attribute is included in the range so that the file is not filtered). Intuitively, however, we believe it is easier to provide an approximate query condition and allowing the search engine to rank all files based on their similarity to the condition than it is to guess at the correct filtering range, which may require overfitting or increasing the range in several query iterations.

Based on the above results, we conclude that our approach of providing flexible query conditions for non-content search dimensions has the potential to considerably improve search accuracy over current filtering approaches. Future work could involve validating this potential in more extensive user studies.

3.2.5 Impact of Multi-dimensional Scoring on Results

To complement the last section, where we studied the ranking of a single target file with respect to a set of related queries, we now consider the impact of our scoring approach on the entire set of top-k files returned in answer to a query. Specifically, we compare

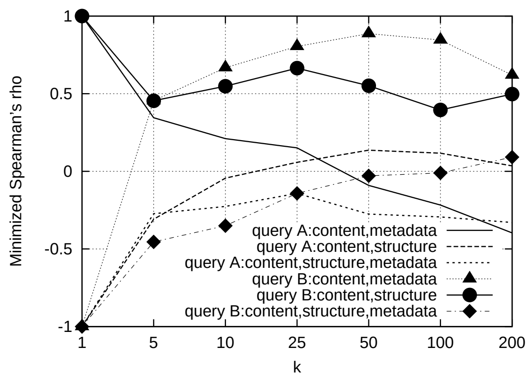


Figure 3.5: ρ_{min} *value for various multi-dimensional queries as a function of* k.

the query results for several multi-dimensional queries with those of a content-only query. To measure the impact of our techniques, we use the *minimized Spearman's rho* as described in [42]. The standard Spearman's rho (ρ) measures the distance between l_1 and l_2, two permutations of the same list. The minimized Spearman's rho (ρ_{min}) is an adaptation of the standard Spearman's rho to top-k lists, which may not overlap. We normalize the minimized Spearman's rho between -1 and 1, where a score of -1 means that objects in the two top-k lists are disjoint, and a score of 1 means the two lists are identical:

$$\rho_{min} = 1 - \frac{6 * \sum d_i^2}{k(k+1)(2k+1)}$$

where k is the number of results returned, d_i is the difference in rank between each object that appears in l_1 or l_2; an object that does not appear in one of the list is considered to have a rank of $k+1$ in that list.

Figure 3.5 shows the ρ_{min} values for various multi-dimensional queries as a function of k. We use two different queries, A and B, to which we add dimension conditions. For Query B we see that the addition of either metadata or structure conditions has only a slight effect on the overall results (indicated by the respective lines staying above 0.5). The combination of both, however, results in significant changes to the set of results. In contrast, for Query A the addition of the metadata dimension provides us with a spearman score ranging from 1.0 to -0.4 indicating that as we increase k the results change significantly. This indicates that the set of files relevant to the content condition of the query and the meta-data condition are quite different. The addition of the structural condition lessens this trend.

Our results show that the multi-dimensional scoring modifies the top-k results with the impact being the most visible for smaller values of k. We have also shown that the degree of change is dependent on the conditions with which the query is extended. Set of conditions whose relevant files are similar will result in very little movement, or introduction of new results, into the final top-k files.

3.3 Summary

In this chapter, we presented a unified scoring framework for multi-dimensional queries over personal information file systems. We proposed individual *IDF*-based scoring approaches for content, metadata, and structure queries. In particular, we defined structure and metadata relaxation tools for our scenario. Our scoring approaches take into account the number of files that match a particular query condition (or relaxation of that condition) to assign scores to each query dimensions. Individual dimension scores can then easily be aggregated.

We implemented and evaluated our scoring framework as part of the Wayfinder file system. Our evaluation has shown that our *IDF*-based scoring approach provides a meaningful distribution of scores that captures the specificity of each dimension. Additionally, we have shown that our multi-dimensional score aggregation technique preserves the properties of individual dimension scores and has the potential to significantly improve ranking accuracy. We reported on the impact of our multi-dimensional scoring on query answers.

Chapter 4

Efficient Query Processing

In Chapter 3, we presented a scoring framework that considers relaxed query conditions on several query dimensions. In these efforts, we performed a qualitative evaluation, which shows that allowing for some flexibility in the structural and metadata components of the query in addition to flexibility in the textual component significantly increases the usefulness of search answers. Our approach (Section 3.1) combines the content, metadata, and structure dimensions in a unified scoring framework using an *IDF*-based interpretation of scores for each dimension.

Efficiently supporting the relaxed queries and scoring techniques presented in Chapter 3, however, requires new indexing structures and algorithms designed to handle query relaxations. We have adapted techniques from the IR literature to search on the content dimension. For the metadata dimension, efficient indexes are proposed to identify and score query answers. The structure dimension, which exhibits complex relaxations such as permutations, is particularly challenging because:

- The index structure for the structure dimension is query dependent. It is not practical to construct a query-independent index of the file system because this would entail enumerating all possible relaxations of all directory pathnames, making efficient query-dependent index construction a critical issue.
- The query relaxation indexes grow exponentially with the size of the query condition. The index structures and algorithms should scale to handle a possibly large number of approximate matches.

Thus, in this chapter, we focus on the efficiency of our approach and present new data structures and index construction optimizations to address the above mentioned challenges. Our techniques and data structures work in conjunction with our adaptation of existing top-k algorithms to provide an efficient implementation of our overall framework.

The rest of the chapter is organized as follows. In Section 4.1, we discuss our choice of top-k query processing algorithm and present our novel static indexing structures and score evaluation techniques. Section 4.2 describes the dynamic indexing structures and index construction algorithms for structural relaxations. Finally, we present our experimental results in Section 4.3.

4.1 Efficient Top-K Query Processing

While Chapter 3 focuses on the flexible multi-dimensional scoring model and its impact on search result quality, the focus of this chapter is on the efficient evaluation of flexible multi-dimensional queries. In this section, we describe the different components of our query processing implementation.

We have adapted the Threshold Algorithm (TA) [43] to our scenario. *TA* uses a threshold condition to avoid evaluating all possible matches to a query, focusing instead on identifying the k best answers.

TA takes as input several sorted lists, each containing the system's objects (files in our scenario) sorted in descending order according to their relevance scores for a particular attribute (dimension in our scenario), and dynamically accesses the sorted lists until the threshold condition is met to find the k best answers without evaluating all possible matches to a query. In our adaptation, TA uses the aggregation approach described in Section 3.1.5 to compute the unified relevance score for each candidate object that it considers.

Critically, *TA* relies on *sorted* and *random accesses* to retrieve individual attribute scores. Sorted accesses, that is, accesses to the sorted lists mentioned above, require the files to be returned in descending order of their scores for a particular dimension. Random accesses require the computation of a score for a particular dimension for any given file. Random accesses occur when TA chooses a file from a particular list corresponding to some dimension, then needs the scores for the file in all the other dimensions to compute its unified score. To use *TA* in our scenario, our indexing structures and algorithms need to support both sorted and random access for each of the three dimensions.

We now present our efficient indexing and scoring techniques for multi-dimensional approximate query processing on each dimension: content (Section 4.1.1), metadata (Section 4.1.2), and structure (Section 4.1.3). In addition, we briefly present access methods to support both sorted and random access on the content (Section 4.1.1) and metadata (Section 4.1.2) dimensions. Note that while very relevant to this section, the work for context and metadata dimensions are done outside the context of this thesis. We will then expand on the specificities of the structure dimension and the need for dynamic index structures to support sorted and random accesses on structure relaxation scores in Section 4.2.

4.1.1 Evaluating Content Scores

As mentioned in Section 3.1.2, we use existing $TF \cdot IDF$ methods to score the content dimension. Random accesses are supported via standard inverted list implementations, where, for each query term, we can easily look up the frequency with which the term appears in the entire file system as well as in a particular file. We support sorted accesses by keeping the inverted lists in sorted order; that is, for the set of files that contain a particular term, we keep the files in sorted order according to their TF scores (normalized by file size) for that term. We then use the TA algorithm recursively to return files in sorted order according to their content scores for queries that contain more than one term.

4.1.2 Evaluating Metadata Scores

Sorted access for a metadata condition is implemented using the appropriate relaxation DAG index. First, exact matches are identified by identifying the deepest DAG node N that matches the given metadata condition (see Section 3.1.3). Once all exact matches have been retrieved from N's leaf descendants, approximate matches are produced by traversing up the DAG to consider more approximate matches. Each parent contains a larger range of values than its children, which ensures that the matches are returned in decreasing order of metadata scores. Similar to the content dimension, we use the TA algorithm recursively to return files in sorted order for queries that contain multiple metadata conditions.

Random accesses for a metadata condition require locating in the appropriate DAG index the closest common ancestor of the deepest node that matches the condition and the deepest node that matches the file's metadata attribute (see Section 3.1.3). This is implemented as a simple DAG traversal algorithm. Random accesses for queries with multiple metadata conditions require the traversal of the appropriate DAG index for each of the metadata conditions.

4.1.3 Evaluating Structure Scores

In this section, we focus on the problem of assigning a structure score to a file when given a (possibly relaxed) structure query path. For a given query, the structure score of a file depends on the directory in which the file is stored and how close the directory matches the query condition. All files within the same directory will therefore have the same structure score.

To compute the structure score of a file f in a directory d that matches the (exact or relaxed) structure condition P of a given query, we first have to identify all the directory paths, including d, that match P. (For the rest of the section, we will use *structure condition* to refer to the original condition of a particular query and *query path* to refer to a possibly relaxed form of the original condition.) For each such directory, we count the number of files in the directory, add these to get the total number of files matching P and compute the structure score of these files for the query using Definition 2 of Chapter 3. The score computation step itself is straightforward; the complexity resides in the directory matching step. Node inversions complicate matching query paths with directories, as all possible permutations have to be considered. Specific techniques and their supporting index structures need to be developed.

Several techniques for XML query processing have focused on path matching. Most

Algorithm 2 *matchDirectory*(Q)

```
1.  $componentList \Leftarrow extractComponents(Q)$
2.  $DirListSet \Leftarrow \{\}$
3.  **for** each $component$ in $componentList$ **do**
4.      $DirList \Leftarrow getDirPositionPairs(component)$
        {$getDirPositionPairs$ queries the inverted index and returns a list of pairs for
        $component$.}
5.      $DirListSet \Leftarrow DirListSet \cup \{DirList\}$
6.  $DirIDSet \Leftarrow \cap\{DirIDs(D) | D \in DirListSet\}$
7.  $relationshipSet \Leftarrow extractRelationships(Q)$
8.  $DirMatches \Leftarrow \{\}$
9.  **for** each $id \in DirIDSet$ **do**
10.     $pairs \Leftarrow extractPairs(id, DirListSet)$
        {For a given $id$, $extractPairs$ extracts pairs containing $id$ from lists in $DirListSet$}
11.     $positions \Leftarrow extractPositions(pairs)$
12.     $isAMatch \Leftarrow true$
13.     **for** each $r \in relationshipSet$ **do**
14.         **if** not $matchRelationship(r, positions)$ **then**
15.             $isAMatch \Leftarrow false$
16.     **if** $isAMatch$ **then**
17.         $DirMatches \Leftarrow DirMatches \cup \{getDir(id)\}$
18. **return** files in directories of $DirMatches$.
```

notably, the *PathStack* algorithm [18] iterates through possible matches using stacks, each corresponding to a query path component in a fixed order. To match a query path that allows permutations (because of node inversion) for some of its components, we need to consider all possible permutations of these components (and their corresponding stacks) and a directory match for a node group may start and end with any one of the node group components, which makes it complicated to adapt the *PathStack* approach to our scenario.

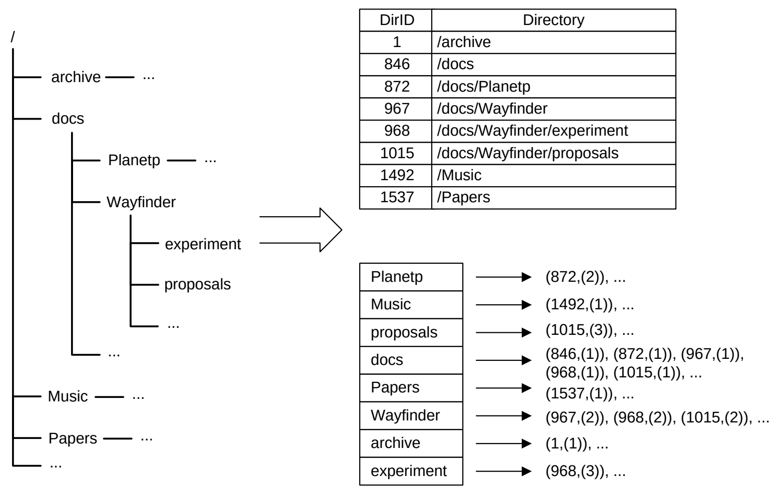
We use a two-phase algorithm (Algorithm 2) to identify all directories that match a query path. First, we identify a set of candidate directories using the observation that for a directory d to match a query path P, it is necessary for all the components in P to appear in d (line 1-6). For example, the directory */docs/proposals/final/Wayfinder* is a potential match for the query path */docs/(Wayfinder//proposals)* since the directory contains all three components *docs*, *Wayfinder*, and *proposals*. We implement an inverted index mapping components to directories to support this step (see Figure 4.1).

We extend our index to maintain the position that each component appears within each directory (Figure 4.1). For efficiency, each directory is represented by an integer directory id, so that each entry in the index is a pair of integers ($DirID$, *position*).[1] This augmented index allows us to quickly check structural relationships between components of a directory. For example, a parent-child relationship between two components n_1, $(DirID_1, Pos_1)$, and n_2, $(DirID_2, Pos_2)$, can be verified by checking that $DirID_1 = DirID_2$ and $Pos_2 = Pos_1 + 1$. This is similar to some XML labeling techniques [64]; however, our approach separates the structural relationship information from the node (directory) labeling, allowing us to assign immutable and persistent ids to directories.

In the second phase, we extract from the query path: (1) the set of node groups representing possible permutations of components, and (2) a sequence of logical conditions representing the left to right parent-child or ancestor-descendant relationship between each component-component or component-node group pairs (line 7,10). For example, we would extract the node group *(Wayfinder//proposals)* and the sequence (*/docs*, *docs/(Wayfinder//proposals)*) from the query path */docs/(Wayfinder//proposals)*. Then, to compute whether a directory matches a query path, we would first identify parts of the directory that match the node groups (line 11). Finally, we would attempt to find an ordering of components and node groups that would match the generated sequence of conditions (line 12-15). If we can find such an ordering, then the directory matches the query path; otherwise, it does not (line 16-17).

Given the above index, suppose that we want to compute whether the candidate directory */docs/proposals/final/Wayfinder* matches the query path */docs/(Wayfinder //proposals)*. The index would tell us that /, *docs*, *Wayfinder*, and *proposals* appear at positions 0, 1, 4, and 2, respectively. We would then compute that the components *proposals* and *Wayfinder* appearing at positions 4 and 2 represents a valid match for the node group *(Wayfinder//proposals)* of the query path; we say that this node group component spans positions 2-4 for the candidate directory. We then compute that the

[1]Directories containing multiple components with the same name such as */a/.../a/..* would contain multiple pairs for the repeating component. Our matching algorithm correctly accounts for such repetitions.



DirID	Directory
1	/archive
846	/docs
872	/docs/Planetp
967	/docs/Wayfinder
968	/docs/Wayfinder/experiment
1015	/docs/Wayfinder/proposals
1492	/Music
1537	/Papers

Planetp	(872,(2)), ...
Music	(1492,(1)), ...
proposals	(1015,(3)), ...
docs	(846,(1)), (872,(1)), (967,(1)), (968,(1)), (1015,(1)), ...
Papers	(1537,(1)), ...
Wayfinder	(967,(2)), (968,(2)), (1015,(2)), ...
archive	(1,(1)), ...
experiment	(968,(3)), ...

Figure 4.1: A directory tree and its structure index.

ordering 0, 1, (2-4) of */*, *docs*, *(Wayfinder//proposals)* satisfies the left-to-right relationships extracted for the query path and thus concludes that the candidate directory is a valid match for the query path.

Obviously, we also need to be able to efficiently find the files residing in any given directory to support scoring. The file system itself supports this functionality.

4.2 Dynamic Evaluation of Structure Relaxations

The structure dimension scoring strategy presented in the previous section assumes knowledge of the matching query path, i.e., takes as input an (exact or relaxed) query path to assign structure scores to file. In a scenario that allows for relaxed structure query conditions, it is necessary to identify the query paths relaxations. Building a static index that would hold all possible relaxation query paths is not a realistic option as this would entail enumerating all possible query combinations, a prohibitively expensive task. This raises the need for efficient dynamic index computation at query time. In this section, we discuss the building of such indexes, as well as sorted and random access methods to efficiently access them.

We now present our dynamic index structures and algorithms for querying the structure dimension. This dimension brings the following new challenges:

- The DAG structures we use to represent query relaxations (Section 3.1.4.3) are query-dependent, and their size grows exponentially with the query size, i.e., the number of path nodes in the query. As such they must be dynamically built at query time, and so efficient index building and traversal techniques are critical issues.
- The *TA* algorithm requires efficient sorted and random access to the single-dimension scores (Section 4.1). In particular, random accesses can be very expensive. We need efficient indexes and traversal algorithms that support both types of access.

We propose the following techniques and algorithms to address the above challenges. We incrementally build the query dependent DAG structures at query time, only materializing those DAG nodes necessary to answer a query (Section 4.2.1). To improve sorted access efficiency, we propose techniques to skip the scoring of unneeded DAG nodes by taking advantage of the containment property of the DAG (Section 4.2.2). We improve random accesses using a novel algorithm that efficiently locates and evaluates only the parts of the DAG that match the file requested by each random access (Section 4.2.3).

Our techniques and algorithms result in fast computation of the structure dimension scores via either sorted or random access.

4.2.1 Incremental Identification of Relaxed Matches

As discussed in Chapter 3, we represent all possible relaxations of a query condition, along with the corresponding *IDF* scores for (files that match) each relaxation, using a DAG structure (see Figure 3.2 for an example). The DAG is created by incrementally applying query relaxations to the original query condition. Children nodes of a DAG node are more relaxed versions of the query condition and therefore match at least as many answers as their parents (containment property). The *IDF* score associated with a DAG node can be no greater than the score associated with its parents.

Scoring an entire query relaxation DAG can be expensive as they grow exponentially with the size of the query condition. For example, there are 5, 21, 94, 427, and 1946 nodes in the respective complete DAGs for query conditions $/a$, $/a/b$, $/a/b/c$, $/a/b/c/d$, $/a/b/c/d/e$. However, in many cases, enough query matches will be found near the top of the DAG, and a large portion of the DAG will not be scored. Thus, we use a lazy evaluation approach to incrementally build the DAG, expanding and scoring DAG nodes to produce additional matches when needed. The partial evaluation should nevertheless ensures that directories (and therefore files) are returned in the order of their scores.

Our lazy evaluation approach is detailed in the $IncrementalDAG$ algorithm shown in Algorithm 3. The algorithm is based on a look-ahead principle, where all the children nodes of expanded nodes (starting with the root node) are materialized, scored, and added into a priority queue based on their scores. The function *applyRelaxations*, which creates children nodes by applying query relaxations, is detailed in Algorithm 1 in Chapter 3. In a greedy fashion, the unexplored DAG node with the highest score is the next one to be expanded, and files matching the corresponding relaxed query are returned. Node expansion continues until we have retrieved enough matches. Since the DAG definition ensures that children nodes have scores that are no greater than that of their parents, the algorithm is guaranteed to return matches in the order of their scores.

For a simple top-k evaluation on the structure condition, our lazy DAG building algorithm is applied and stops when k matches are identified. For complex queries involving multiple dimensions, the algorithm can be used for sorted access to the structure condition. Random accesses are more problematic as they may access any node in the DAG. The DAG building algorithm can be used for random access, but any random access may lead to the materialization and scoring of a large part of the DAG.[2] In the next sections we will discuss techniques to optimize sorted and random access to the query relaxation DAG.

[2]We could modify the algorithm to only score the node that actually matches the file of a random access. However, with our index, scoring is cheaper than computing whether a specific file matches a given node.

Algorithm 3 *IncrementalDAG*$(root, k)$

```
1.  $Queue \Leftarrow PriorityQueue(getScore(root), root)$
2.  $topKFiles \Leftarrow \{\}$
3.  $currentNode \Leftarrow nil$
4.  $seenNodes \Leftarrow \{root\}$
5.  while $notEmpty(Queue)$ do
6.    if $currentNode \neq nil$ then
7.      $childNodes \Leftarrow applyRelaxations(currentNode)$
        {$applyRelaxations$ applies structural relaxations to the query condition of
        $currentNode$ and returns DAG nodes for resulting relaxed query conditions}
8.      for each $n \in childNodes$ do
9.        if $n \notin seenNodes$ then
10.         $addNode(Queue, getScore(n), n)$
11.         $seenNodes \Leftarrow seenNodes \cup \{n\}$
12.   $currentNode \Leftarrow removeNode(Queue)$
13.   $fileMatches \Leftarrow matchDirectory(getQuery(currentNode))$ {$matchDirectory$ is de-
      scribed detailed in Section 4.1.3.}
14.   $topKFiles \Leftarrow topKFiles \cup fileMatches$
15.   if $|topKFiles| \geq k$ then
16.     return $topKFiles$
17. return $topKFiles$
```

4.2.2 Improving Sorted Accesses

Evaluating queries with structure conditions using the lazy DAG building algorithm can lead to significant query evaluation times as it is common for multi-dimensional top-k processing to access very relaxed structure matches, i.e., matches to relaxed query paths that lay at the bottom of the DAG, to compute the top-k answers.

In Chapter 3, we pointed out that not every possible relaxation leads to the discovery of new matches. For example, in Figure 3.2, the query paths */docs/Wayfinder/proposals*, *//docs/Wayfinder/proposals*, and *//docs//Wayfinder/proposals* have exactly the same scores of 1, which means that no additional files were retrieved after relaxing */docs /Wayfinder/proposals* to either *//docs/Wayfinder/proposals* or *//docs//Wayfinder/proposals* (Definition 2 in Chapter 3). By extension, if two DAG nodes share the same score, then all the nodes in the paths between the two DAG nodes must share the same score as well per the DAG definition. This is formalized in Theorem 5.

Theorem 5 *Given the structural* $score_{idf}$ *function defined in Definition 2, if a query path* P' *is a relaxed version of another query path* P*, and* $score_{idf}(P') = score_{idf}(P)$ *in the structure DAG, any node* P'' *on any path from* P *to* P' *has the same structure score*

Algorithm 4 *DAGJump*(*srcNode*)

```
1.  $s \Leftarrow getScore(srcNode)$
2.  $currentNode \Leftarrow srcNode$
3.  **loop**
4.    $targetDepth \Leftarrow getDepth(currentNode)$
5.    $childNode \Leftarrow firstChild(currentNode)$
6.    **if** $getScore(childNode) \neq s$ or
        $hasNoChildNodes(childNode)$ **then**
7.      exit loop
8.    $currentNode \Leftarrow childNode$
9.  **for** each $n$ s.t. $getDepth(n) = targetDepth$ and $getScore(n) = s$ **do**
10.   Evaluate bottom-up from $n$ and identify ancestor node set $S$ s.t. $getScore(m) = s, \forall m \in S$
11.   **for** each $m \in S$ **do**
12.     **for** each $n'$ on path $p \in getPaths(n, m)$ **do**
13.       $setScore(n', s)$
14.       $setSkippable(n')$
15.     **if** $notSkippable(m)$ **then**
16.       $setSkippable(m)$
```

as $score_{idf}(P)$*, and* $F(P') = F(P'') = F(P)$*, where* $F(P)$ *is the set of files matching query path* P*.*

Proof. If $score_{idf}(P') = score_{idf}(P)$, then by definition $N_{P'} = N_P$ (Definition 2). Because of the containment condition, for any node P'' on any path from P to P', we have $F(P') \supseteq F(P'') \supseteq F(P)$ and $N_{P'} \geq N_{P''} \geq N_P$. Thus, $N_{P'} = N_{P''} = N_P$ and $F(P') = F(P'') = F(P)$, since otherwise there exists at least one file which belongs to $F(P')$ (or $F(P'')$) but does not belongs to $F(P)$ and $N_{P'} \neq N_P$ (or $N_{P''} \neq N_P$), contradicting our assumption $N_{P'} = N_P$ (and $N_{P''} = N_P$). ■

Theorem 5 can be used to speed up sorted access processing on the DAG by skipping those DAG nodes that will not contribute to the answer.

We propose Algorithm 4, *DAGJump*, based on the above idea. It includes two steps: (a) starting at a node corresponding to a query path P, the algorithm performs a depth-first traversal and scoring of the DAG until it finds a parent-child pair, P' and *child*(P'), where $score_{idf}(child(P')) < score_{idf}(P)$; and (b) score each node P'' at the same depth (distance from the root) as P'; if $score_{idf}(P'') = score_{idf}(P)$, then traverse all paths from P'' back toward the root; on each path, the traversal will reach

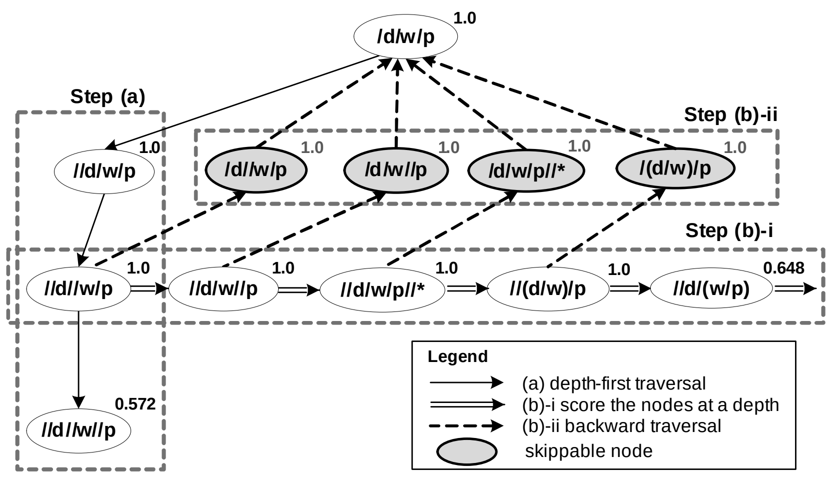


Figure 4.2: An example execution of *DAGJump*. IDF scores are shown at the top right corner of each DAG node.

a previously scored node P^*, where $score_{idf}(P^*) = score_{idf}(P)^3$; all nodes on all paths from P'' to P^* can then be marked as skippable since they all must have the same score as P''.

An example execution of *DAGJump* for our query condition */docs/Wayfinder/proposals* is given in Figure 4.2. The two steps from Algorithm 4 are performed as follows: (a) starting at the root node with a score of 1, *DAGJump* performs a depth-first traversal and scores the DAG nodes until it finds a node that has a smaller score than 1 (*//d//w//p*); and (b) *DAGJump* traverses each node at the same depth as *//d//w/p* (the parent node of *//d//w//p*); for the four such nodes that have a score 1, *DAGJump* marks as skippable all nodes that are on their path to the root node.

It is worth noting that Algorithm 4's depth-first traversal always follows the first child. We experimented several different heuristics for child selection (first child, middle child, last child, and random) and found no significant differences in performance.

The *DAGJump* algorithm is integrated into our lazy DAG building algorithm to reduce the processing time of sorted accesses.

[3]This condition is guaranteed to occur because of our DAG expansion Algorithm 3.

4.2.3 Improving Random Accesses

Top-k query processing requires random accesses to the DAG. Using sorted access to emulate random access tends to be very inefficient as it is likely the top-k algorithm will access a file that is in a directory that only matches a very relaxed version of the structure condition, resulting in most of the DAG being materialized and scored. While the *DAGJump* algorithm somewhat alleviates this problem by reducing the number of nodes that need to be scored, efficient random access remains a critical problem for efficient top-k evaluations.

We present the *RandomDAG* algorithm (Algorithm 5) to optimize random accesses over our structure DAG. The key idea behind *RandomDAG* is to skip to a node P in the DAG that is either a close ancestor of the actual least relaxed node P' that matches the random access file's parent (containing) directory d or P' itself and only materialize and score the sub-DAG rooted at P as necessary to score P'. The intuition is that we can identify P by comparing d and the original query condition. In particular, we compute the intersection between the query condition's components and d. P is then computed by dropping all components in the query condition that is not in the intersection, replacing parent-child with ancestor-descendant relationships as necessary. The computed P is then guaranteed to be equal to or an ancestor of P'. As DAG nodes are scored, the score together with matching directories are cached to speed up future random accesses.

As an example, for our query condition */docs/Wayfinder/proposals* in Figure 3.2, if the top-k algorithm wants to perform a random access to evaluate the structure score of a file that is in the directory */archive/proposals/Planetp*, *RandomDAG* will first compute the close ancestor to the node that matches */archive/proposals/Planetp* as the intersection between the query condition and the file directory, i.e., *//proposals*, and will jump to the sub-DAG rooted at this node. The file's directory does not match this query path, but does match its child *//proposals//** with a structure score of 0.197. This is illustrated in Figure 4.3 which shows the parts of the DAG from Figure 3.2 that would need to be accessed for a random access to the score of a file that is in the

Algorithm 5 *RandomDAG(root, DAG, F)*

```
1.  $p \Leftarrow getDirPath(F)$
2.  **if** $p \in DAGCache$ **then**
3.    **return** $getScoreFromCache(DAGCache, p)$
4.  $droppedComponents \Leftarrow$
        $extractComponents(root) - extractComponents(p)$
5.  $p' \Leftarrow root$
6.  **for** each $component \in droppedComponents$ **do**
7.    $p' \Leftarrow nodeDeletion(p', component)$
8.  **loop**
9.    $n \Leftarrow getNextNodeFromDAG(p')$
      {$getNextNodeFromDAG$ incrementally builds a sub-DAG rooted at $p'$ and returns the
      next DAG node in decreasing order of scores.}
10.   $fileMatches \Leftarrow matchDirectory(getQuery(n))$
11.   $dirPaths \Leftarrow getDirPaths(fileMatches)$
12.   $addToCache(DAGCache, dirPaths, getScore(n))$
13.   **if** $p \in dirPaths$ **then**
14.     **return** $getScore(n)$
```

directory */archive/proposals/Planetp.*

We now prove the correctness of Algorithm 5.

Lemma 1 *Let p (containing a set S_1 of terms) be a full pathname. Then, within a structure DAG rooted at a path query Q (containing a set S_2 of terms), any query Q' (other than the match-all query //*) that is matched by p must contain and only contain terms from the set $S_1 \cap S_2$. The match-all query node $*$ is not counted as a term.*

Proof. We prove it by contradiction. Assume the conclusion is not true. A path query Q' (containing a set S_3 of terms) other than $//*$ is matched by p and Q' has at least one term t s.t. $t \in S_3$ and $t \notin S_1 \cap S_2$. Based on the definition of matches to a path query, if p matches Q' it must contain term t. Therefore, $t \in S_1$, which implies $t \notin S_2$.

Since Q' is in the DAG rooted at Q, Q' is a relaxed version of Q. As none of structure relaxation rules introduces new terms, Q' must only contain terms from S_2. We have $t \in S_3 \subseteq S_2$, contradicting $t \notin S_2$. ■

Lemma 2 *Within a structure DAG rooted at a path query Q (containing a set S_1 of terms), there is a least relaxed query Q' among all queries only containing terms from a set S_2. We can get Q' by applying a series of node deletions to Q, each involving a term in $S_1 - S_2$.*

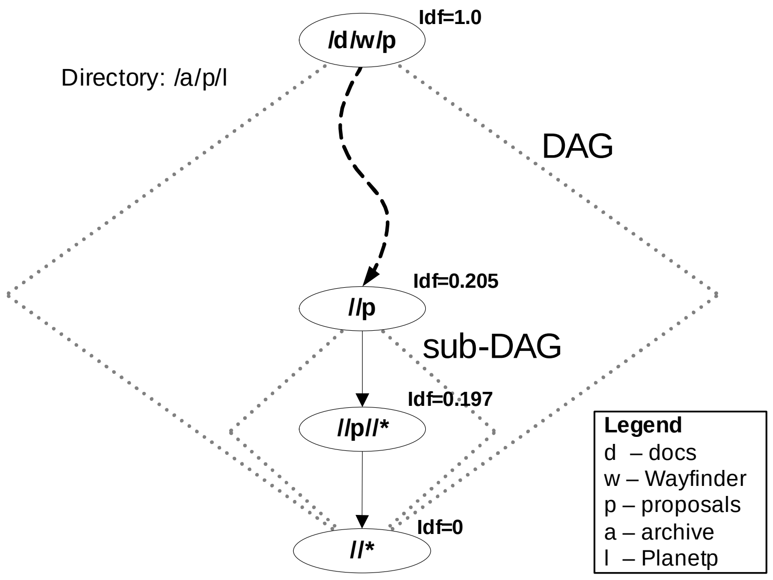


Figure 4.3: An example execution of *RandomDAG* that returns the score of a file that is in the directory */archive/proposals/Planetp* for the query condition */docs/Wayfinder/proposals*.

Proof. First, when a series of node deletions are applied to a path query, the final relaxed query is independent of the order of node deletion operations. This is true since each node deletion operation removes a query node, which only affecting its adjacent nodes. The operation does not change the order of other query nodes.

Second, if a node deletion follows a series of non-node-deletion operations, the result query is a relaxed version of the query that is obtained by applying the same node deletion to the original query. This can be derived from the containment condition and the definition of node deletion. By induction it is easy to see when multiple node deletion operations and non-node-deletion operations interleave with each other, the result query is a relaxed version of the query that is obtained by applying the same node deletion operations to the original query.

Last, within the DAG any query Q'' only containing terms from S_2 is obtained by applying a series of relaxations that can be separated into a set of node deletion operations, noted O_{ND}, and a set of non-node-deletion operations, noted $O_{\neg ND}$. Based on the above conclusions, Q'' is a relaxed version of another query Q' that is obtained by applying node deletion operations O_{ND} to Q, and Q' is unique since it is independent of the order of these operations. Therefore, Q' is the least relaxed query among all queries only containing terms from S_2. As node deletion is the only operation that can remove terms that are not in S_2, each operation in O_{ND} must involve a term in $S_1 - S_2$. ■

It is easy to see the correctness of Theorem 6 based on Lemma 1 and Lemma 2. This proves the correctness of Algorithm 5.

Theorem 6 *Let f be a file that has a full pathname p and f matches a relaxed query Q'' in a structure DAG rooted at a path query Q. Q'' must be a descendant node of Q' that is obtained by applying node deletion operations to Q, each involves a term in Q but not in p.*

4.3 Experimental Results

We now turn to evaluating the search performance achievable using the indexing structures, scoring algorithms, and top-k algorithm described in Sections 4.1 and 4.2. In particular, we report query performance for a large set of queries against a relatively large personal data set. We show that the optimizations described in Section 4.2 help to significantly reduce the query processing times, making relaxed multi-dimensional search quite practical to use. We also report the space overheads imposed by the persistent storage of our indexes. Finally, we briefly consider the scalability of our system against the size of the data set as well as increasing values of k.

4.3.1 Experimental Setup

Experimental environment.

The platform we use is same as the one that has been used in Chapter 3. We evaluate the performance of our search approach by measuring the query processing time of the same prototype that is described in Chapter 3. Reported query processing times are averages of 40 runs, after 40 warm-up runs to avoid measurement JIT effects.

All caches (except for any Berkeley DB internal caches) are flushed at the beginning of each run.

Data set.

We use the same data set that has been used in Chapter 3.

Query set.

In the absence of meaningful benchmarks, we construct a large set of synthetic queries (80) to evaluate our system. These queries are designed to include a large variety of query conditions combined in different ways across the three dimensions. They also target files from three different content categories, including email (20 queries), document (40 queries for ebooks, academic papers, etc.), and media files (20 queries for music, etc.). This large and diverse set of queries allow us to explore the performance of our system across the parameter space that should include most real-world search scenarios.

As noted in [80], people often know exactly what they are looking for when they execute searches for an email message, a file, or even a Web page. Thus, each of our query targets a specific file f, and so is built using (relaxations of) f's attributes. The query conditions are formed as follows.

- *Content:* Each condition has 2 to 4 terms chosen randomly from f's content.
- *Metadata:* Each date (last modified) is randomly chosen from a small range (± 7 days to represent cases where users are searching for files they recently worked on) or a large range (± 3 months to represent cases where users are searching for files that they have not worked on for a while and so only vaguely remember the last modified times) around f's actual last modified date. Each file type (extension) is randomly chosen from `.txt` or `.pdf` for a document; otherwise, it is the correct file type.
- *Structure:* Each condition is randomly chosen from: (a) the correct path p, (b) one random word was dropped from p, (c) two adjacent words in p are swapped, and (d) one word in p was misspelled.

Category	No. of Queries	$k = 10$		$k = 20$	
		Recall	MRR	Recall	MRR
email	20	0.85	0.48	0.90	0.49
document	40	0.88	0.69	0.98	0.70
media	20	0.85	0.64	1.00	0.65

Table 4.1: The mean recall (Recall) and mean reciprocal rank (MRR) of the target file for each query category.

Choosing k. Query performance is a function of k, the number of top ranked results that should be returned to the user. We considered two factors in choosing a k value: (1) the mean recall, which, in our case, indicates the percentage of time that the target file is returned as one of the k answers, and (2) the likelihood that users would actually look through all k returned answers for the target file.

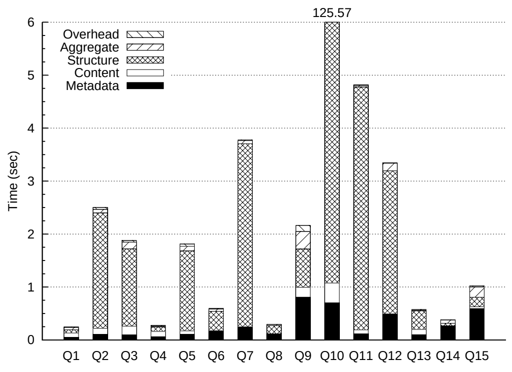
Table 4.1 shows the mean recall (and mean reciprocal rank to give an idea of the ranking of the target files) for $k = 10$ and $k = 20$. Based on these results, we chose $k = 10$ because $k = 20$ would only increase recall slightly; further, this increase is likely to be of minimal value because experience from Web search has shown that users rarely look beyond the top 10 results. (In fact, reporting search times for $k = 20$ would have magnified the importance of our optimizations without significantly increasing overall optimized performance results–see Section 4.3.6.)

4.3.2 Base Case Query Processing Performance

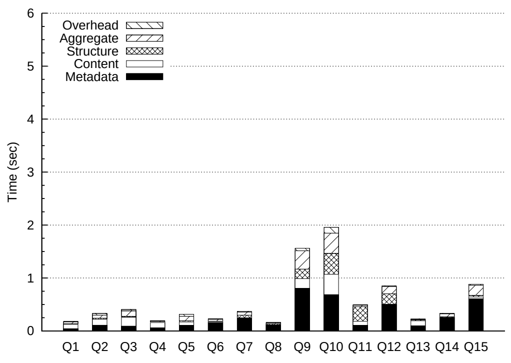
We first report query processing times for the base case where the system constructs and evaluates a structural DAG sequentially without incorporating the DAGJump (Section 4.2.2) and RandomDAG (Section 4.2.3) optimization algorithms. As already mentioned, the k parameter of the top-k algorithm is set to 10 in all cases.

Note that the base case includes the implementation of the matchDirectory (Section 4.1.3) and IncrementalDAG (Section 4.2.1) techniques. These two techniques indeed provide information that is necessary to derive the structure scores of query matches. In particular, the query-dependent DAG is needed to identify the valid relaxations to the structural condition of the query. Without this runtime index, we would not be able to evaluate the structure dimension scores.

Figure 4.4(a) presents the query processing times, including breakdowns of the times



(a) Base Case



(b) DAGJump+RandomDAG

Figure 4.4: Breakdowns of query processing times for 15 queries for (a) the base case, and (b) DAGJump+RandomDAG case.

required for top-k sorted and random accesses for each of the three search dimensions (Metadata, Content, and Structure), top-k aggregation processing (Aggregate), and any remaining overhead (Overhead), for 15 representative queries (5 from each of the three categories). We observe that query processing times can be quite large; for example, query Q10 took 125.57 seconds. In fact, to view the distribution of processing times over all 80 queries, Figure 4.5 plots the cumulative distribution function (CDF) of the processing times, where each data point on the curve corresponds to the percent of queries (Y-axis) that finished within a given amount of time (X-axis). The CDF shows that, for the base case, over 28% of the queries took longer than 2 seconds and 9% took longer than 8 seconds to complete.

We also observe that the processing times for the structure dimension dominate the query processing times for the slowest queries—see the break down of processing times for Q2, Q3, Q5, Q7, Q9, Q10, Q11, and Q12 in Figure 4.4. For the entire set of 80 queries, processing times in the structure dimension comprised at least 63% of the overall query processing times for the 21 slowest queries (those with processing times $\geq$ 2.17 seconds).

Thus, we conclude that it is necessary to optimize query processing to make multi-dimensional search more practical for everyday usage. More specifically, our results demonstrate the need to optimize query processing for the structure dimension as it dominates the overall query processing time for the worst performing queries.

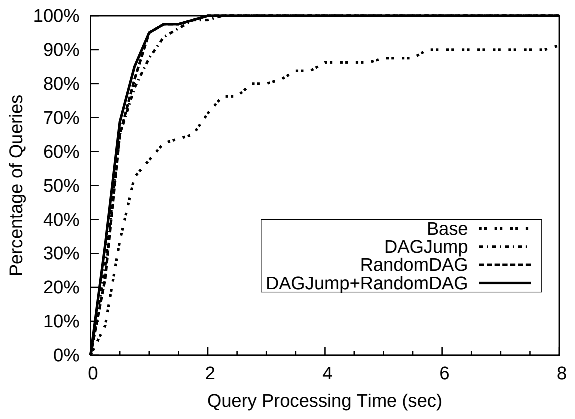


Figure 4.5: The CDF of query processing time for four cases. Most part of RandomDAG and DAGJump+RandomDAG curves overlap with each other.

4.3.3 Query Processing Performance with Optimizations

We now explore the impact of the two optimization algorithms developed in Section 3. Figure 4.4(b) gives the query processing times for the same 15 queries shown in Figure 4.4(a), but after we have applied both the DAGJump and the RandomDAG algorithms.

We observe that these optimizations significantly reduce the query processing times for most of these queries. In particular, the query processing time of the slowest query, Q10, decreased from 125.57 to 1.96 seconds. Although not shown here, all 80 queries now require at most 0.39 seconds of processing time for the structure dimension, and, on average, the percentage of processing time spent in the structure dimension is comparable to that spent in the metadata dimension.

To view the overall effects of the optimizations on all queries, Figure 4.5 plots the CDF of the processing times for all 80 queries for four cases: base case, use of DAGJump, use of RandomDAG, and use of both the DAGJump and RandomDAG optimizations. We observe that, together, the two optimizations remove much of the "tail" of the base case's CDF; the maximum query processing time is below 2 seconds in the DAGJump+RandomDAG curve. This is particularly important because high variance in response time is known to significantly degrade the user-perceived usability of a system. Further, approximately 70% of the queries complete within 0.5 second,

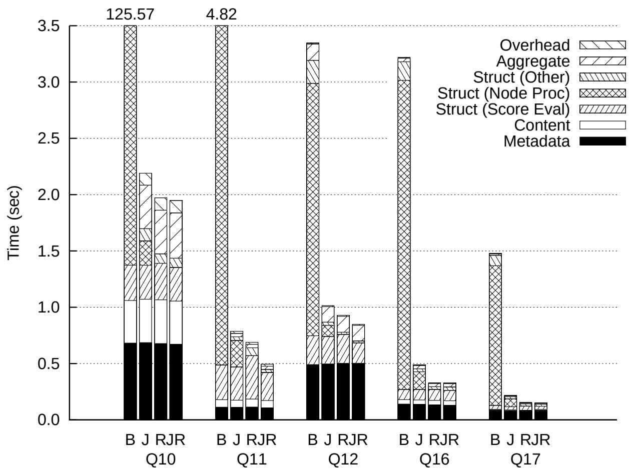


Figure 4.6: Query time breakdown for five representative queries with different impact from various optimizations. B, J, R, JR denote base, DAGJump, RandomDAG, and DAGJump+RandomDAG cases respectively.

while 95% of the queries complete within 1 second; these results show that the two optimizations have made our system practical for everyday usage. This is especially true considering that our prototype has not been optimized for any dimensions other than structure, and, overall, has not been code optimized.

4.3.4 Understanding the Impact of DAGJump and RandomDAG

Interestingly, just using RandomDAG alone achieves much of the benefits of using both optimizations together (Figure 4.5). Using DAGJump alone is almost as good. In this subsection, we study the detail workings of the two optimizations for several queries to better understand the overlap and the differences between the two optimizations.

Figure 4.6 shows break downs of query processing times for five queries (three from Figure 4.4), where the structure processing times have further been broken down into three categories: the node processing time required to check whether a given directory is a member of the set of directories matching a DAG node for random accesses (Node Proc), the time for scoring nodes–most of which is due to sorted accesses–(Score Eval), and other sources of overheads in processing the structure dimension (Other).

We observe that the node processing times typically dominate, which explains the effectiveness of the RandomDAG optimization. In the absence of RandomDAG, DAGJump also significantly reduces the node processing times because it skips many nodes that would otherwise have to be processed during random accesses. DAGJump is not as efficient as RandomDAG, however, because it cannot skip as many nodes. With respect to sorted accesses, DAGJump often skips nodes that are relatively cheap to score (e.g., those with few matching directories). Thus, it only minimally reduces the cost of sorted accesses in most cases. There are cases such as Q11 and Q12, however, where the cost of scoring nodes skipped by DAGJump is sufficiently expensive so that using RandomDAG+DAGJump is better than just using RandomDAG.

To summarize, our DAGJump algorithm improves query performance when (a) there are many skippable nodes which otherwise would have to be scored during the top-k sorted accesses, and (b) the total processing time spent on these nodes is significant. The RandomDAG algorithm improves query performance when (a) the top-k evaluation requests many random access, and (b) the total processing time that would have been spent on nodes successfully skipped by RandomDAG is significant. In general, RandomDAG is almost as good as using both optimizations together but, there are cases where using RandomDAG+DAGJump noticeably further reduces query processing times.

4.3.5 Storage Cost

We report the cumulative size of our static indexes of Section 4.1 to show that our approach is practical with respect to both space (storage cost) and time (query processing performance). In total, our indexes require 246 MB of storage, which is less than 2% of the data set size (14.3 GB). This storage is dominated by the content index, which accounts for almost 92% of the 246 MB. Data for the structure dimension accounts for approximately 1.3% of the total storage with 3.5 MB. The indexes are so compact compared to the data set because of the large sound (music) and video (movie) files. As future data sets will be increasingly media rich, we expect that our indexes will continue to require a relatively insignificant amount of storage.

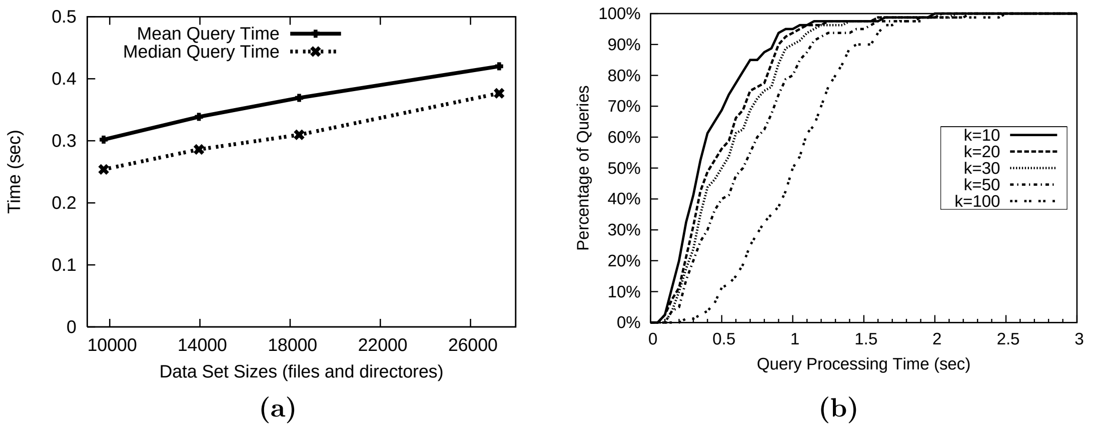


Figure 4.7: (a) The mean and median query times for queries targeting email and documents plotted as a function of data set size. (b) CDF of the total query times for different values of k.

4.3.6 System Scalability

We believe that our experimental data set is sufficiently large that our performance results apply directly to personal information management systems. Nevertheless, we briefly study the scalability of our system to assess its potential to handle very large personal data sets. Figure 4.7(a), which plots average and median query times (for the same set of 80 queries discussed above) against data set size, shows that query performance scales linearly with data set size but with a relatively flat slope (e.g., increase of only 0.1 seconds in mean query processing time when the data set doubles in size). Further, analysis shows that the linear growth is directly attributable to our unoptimized implementation of the top-k algorithm; score evaluation times remain relatively constant vs. data set size. This result is quite promising because there are many known optimizations that we can apply to improve the performance and scalability of the top-k algorithm.

Along a different dimension, we also measured query performance for increasing k values. Results in Figure 4.7(b) show that our approach scales very well with k. For example, the 90th percentile processing time (i.e., the time within which 90% of the queries completed) only increased from 0.87 seconds for $k = 10$ to 0.9 seconds for $k = 20$ to 1.13 seconds for $k = 50$. Average and median query processing times followed the same trend.

4.4 Summary

In this chapter, we have presented indexing structures and dynamic index construction and query processing optimizations to support efficient evaluation of relaxed multi-dimensional searches in personal data management systems. More specifically, we address query processing for relaxed conditions on metadata and structure information. Our approach is uniformly based on building DAG indexes to iteratively access files that match progressively more relaxed approximations of the query conditions. We show how these indexes can be optimized for both sorted and random accesses that are necessary to support a top-k query processing algorithm such as the Threshold Algorithm.

We implemented our proposed approach in the fully functioning file system Wayfinder. We evaluated our implementation by executing a large number of queries against a large, real-life personal data set. Our evaluation shows that our indexes and optimizations are necessary to make multi-dimensional searches efficient enough for practical every-day usage. We also show that our optimized query processing strategies exhibit good behavior across all dimensions, resulting in good overall query performance and good scalability.

Chapter 5

Unified Structure and Content Search

In Chapter 1, we pointed out keyword-only searches do not exploit the rich structural information that is typically available in personal information systems. Using this structural information only as filtering conditions during query processing is too rigid because any mistake in the query will lead to relevant files being missed; a flexible approach that allow for some error in the structural conditions is desired.

Also, the structural heterogeneity complicates the search for specific files. In Example 1 (see Chapter 1), when John tries to find the photos of a Halloween party that was held at home where someone was wearing a witch costume, a content-only search for "Halloween, home, witch", even considering only pictures, is likely to result in many matches, of various relevance. However, none of the pictures *–pic1728.gif, party42.jpg, and IMG_1391.gif–* in the example data set of Figure 1.1 (see Chapter 1) contains all three keywords. *pic1728.gif* and *party42.jpg* contain two of the keywords; their relative rankings would depend on the underlying content scoring function. One of the three picture, *IMG_1391.gif* is however arguably the best match as its directory structure contains the third missing keyword.

Moreover, when searching through their files, users are likely to remember partial structure information about the file. In our example, John believes the photo he is looking for was not in an email attachment, but in his home directory and was annotated with a caption containing the term "Halloween". Based on this information, John can write the following query:

```
[structure=//home[.//caption/"Halloween" and .//"witch"]
```

Current search tools would probably return *IMG_1391.gif* as an exact match to the query but would likely miss returning approximate but relevant matches such as file *party42.jpg* and *pic1728.gif* due to the structure filtering condition (`//home//caption`) and the strict separation between the external (directory structure) and the internal (structure and content) respectively.

Because of the data heterogeneity of personal information systems, we believe it is critical to support approximate matches on both the content and structural components of the query and to allow for query conditions to be evaluated across file boundaries. For this purpose, we propose a unified data model that ignores the traditional physical file boundaries and a unified scoring framework that considers relaxed query conditions on structure and content at once and provide a unified score.

For the rest of the chapter, we first present a unified data and query model to unify the structure outside and inside the files and allow relaxations for the unified structural and content conditions across file boundaries (Section 5.1). We then develop a unified scoring framework to rank answers of our flexible queries (Section 5.2). Next, we present query processing techniques to efficiently assign the relevance scores to query answers (Section 5.3). Last, we evaluate our model and query processing techniques and show our new approach is more flexible and robust than the text search approaches that ignore the structure relationships (Section 5.4).

5.1 Data and Query Model

We now present our model for providing unified flexible search results over personal information. In Section 5.1.1 we discuss our data representation that unifies external and internal structure with content in a XML-like view. We then describe our query model, which allows for approximations in content and structure query conditions in Section 5.1.2.

5.1.1 Unified Data Model

Our data model considers structure both outside and within the file in a unified manner. The whole file system can be seen as an XML document, with the content of files stored in the leaf nodes. We model the whole structure as a rooted, labeled, unordered tree, that contains internal *structure* nodes and leaf *content* nodes. (Empty directory nodes are modeled as having an empty content child node so that all leaf nodes are content nodes.) In the rest of the section we refer to this data representation as the *unified data tree T*.

Definition 7 (Unified Data Tree) *The unified data tree is a rooted, unordered tree, denoted by $T = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ and $\mathcal{E}$ are the sets of nodes and edges of T respectively. Each $n \in \mathcal{N}$ has a label, denoted by $label(n)$[1], and a type, denoted by $type(n)$, of either root, non-root structure, or content node. An edge from node n_i to n_j is denoted by (n_i, n_j).*

Figure 1.1 (Chapter 1) shows a subset of the unified data tree for an example user personal information file system. The external structure (directories) and internal structure (such as the "From" field in an email, or "title" of an ebook file) of files are both represented as internal structure nodes in the unified data tree (the dotted line representing the file boundary is given in the figure for illustration purpose only). Content is stored in the leaves. Abstractly, each leaf node only contains one term although in the implementation, sibling content nodes are collapsed into a single node to save space.

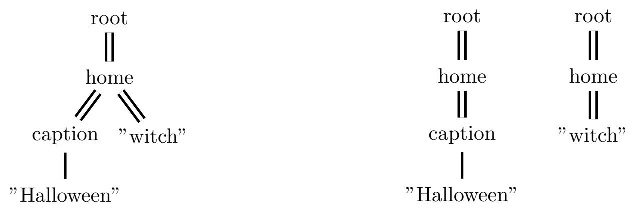
To simplify the discussion, we ignore the file system metadata information such as file size or modification time in this chapter. Metadata can easily be included in the unified structure, with both structural component (e.g., "Last Modified") and leaf node values (e.g., "10/31/09").

5.1.2 Flexible Query Model

Our model allows users to query both the content of the files, using a standard keyword-based model, as well as their structure, internal and/or external. A query over our unified data model is a combination of structural patterns and content terms that can be represented as a twig, in the spirit of XQuery [88].

[1] *For simplicity, we consider "label" to be equivalent to "value" for content nodes in the context of this thesis.*

For example, the twig query given in Example 1 (see Chapter 1) is shown in Figure 5.1(a).



(a) Twig query

(b) Path queries from the decomposition of twig query (a)

Figure 5.1: An example twig query and its decomposition.

In this work, we do not expect users to specify twig queries. Instead, we assume there exists a thoughtfully designed user interface that allows for users to specify query conditions in a convenient form and transforms the user input to twig queries.

5.1.2.1 Query Nodes

As we discussed in Chapter 1, personal data is often very heterogeneous and users could easily make mistakes when providing queries. In such an environment, search tools should be flexible enough to combine and approximate the structure and content information in search queries. To represent this flexibility in our query model, we introduce the following notations for query nodes:

- **Root Node:** The root node, noted *root*, is a query node that is matched by the root of the unified data tree T.
- **Content Node:** A content node, noted *"N"*, is a query node with label N that can be matched only by a leaf node with label N in T.
- **Structure Node:** A structure node, noted N, is a query node with label N that can be matched by any structure (non-leaf) node in T with label N.
- **Generalized Node:** A generalized node, noted $\{N\}$, is a query node with label N that can be matched by either a structure or content node in T with label N.

- **Extended Node:** An extended node, noted *N//**, is a query node that can be matched by the whole subtree rooted at a match to N in T. Content, structure and generalized node can all be extended, but an extended content node is equivalent to the original content node since content nodes only match leaf nodes in T.

- **Path Segment:** A path segment, noted *PS*, is a partial path where each node is either a content, structure or extended node and each edge is either a parent-child edge (/) or an ancestor-descendant edge (//). A path segment PS can be matched by any path P in T where each node in PS is matched by a node in P and the matching nodes in P follows the same edge structure in *PS*.

- **Node Group:** To represent possible permutations of query nodes, we use node groups. They are the same as the node groups introduced in Section 3.1.4.1, except now each node group may contain at most one generalized node (since generalized nodes can be matched by content nodes and each path contains at most one content node), and all edges in the path are ancestor-descendant edges. The placement of the generalized node is fixed at the end of the path although the node labels may permute. Essentially, a node group NG is a query node that can be matched by all paths in T where each path matches a path segment PS that is a valid permutation of NG's components.

  We define the *extension* of a node group ng as the set of all path segments that are contained in ng, each corresponds to a valid permutation of the nodes in ng.

  For example *(home//Halloween//{witch})* is a node group which corresponds to the extension set containing *home//Halloween//{witch}*, *home//witch//{Halloween}*, *Halloween//home//{witch}*, *Halloween//witch//{home}*, *witch//home//{Halloween}* and *witch//Halloween//{home}*.

5.1.2.2 Query Formulation

Our model considers twig query conditions over the unified data tree T. A twig query and its components are defined in regular expressions as follows.

Definition 8 (Path Expression) *A Path Expression PE is of the form:*

$$PE = ((PE_0|PE_1)(//*)? \mid //*)$$

$$PE_0 = (Edge\ (N|NG))^+$$

$$PE_1 = (Edge\ (N|NG))^*\ (Edge\ (\text{“}N\text{”}|\{N\}))$$

where $Edge = (//|/)$.

Definition 9 (Twig Expression) *A Twig Expression TE is of the form:*

$$TE = (PE_0(\text{“[”}\ \text{“.”}(PE|TE)\ (\text{“and”}\ \text{“.”}(PE|TE))^+\ \text{“]”}) \mid PE)$$

Definition 10 (Twig Query) *A Twig Query TQ is of the form:*

$$TQ = root\ TE$$

Intuitively, a twig query starts at the root node of T ($root$) and includes a twig expression. Each twig expression contains a path expression possibly followed by multiple query branches. A path expression consists of any number of structure query nodes (or node groups) connected by parent-child (/) or ancestor-descendant (//) edges. Each query branch is either a path expression or a twig expression. Generalized and content nodes can only be positioned as the last node of the path expression. A path expression can be extended by adding the extension wildcard at the end of the query.

In this chapter, we consider a simplification of the query model that decomposes a twig query into a set of path queries for scoring. A path query is created for each root to leaf path in the twig query.

Definition 11 (Path Query) *A Path Query*[2] *PQ is of the form:*

$$PQ = root\ PE$$

[2]*Like Section 3.1.4.1, for simplicity, the* root *node is optional in the representation of a path query. For example, we consider query /home/Halloween to be equivalent to /root/home/Halloween. This applies to all queries in this chapter.*

Figure 5.1(b) shows the path query decomposition of the twig query from Example 1 (Figure 5.1(a)).

We use path queries as the scoring units and compute the score of a twig query as a function of the scores of the path queries resulting from the twig query decomposition (Section 5.2.2).

5.1.2.3 Query Match

A *match* for a path query PQ in a data tree T is defined as a set of data nodes in T governed by a mapping from query nodes in PQ to data nodes[3] such that the labels match each other and the structural relationships are preserved.

Since a path query PQ is just a specialized rooted tree containing nodes and edges, we let $(\mathcal{N_{PQ}}, \mathcal{E_{PQ}})$ denote PQ, similar to Definition 7 (see Section 5.1.1), and define a match for PQ as follows.

Definition 12 (Path Match) *A* match *for a path query* $PQ = (\mathcal{N_{PQ}}, \mathcal{E_{PQ}})$ *in the unified data tree* $T = (\mathcal{N_T}, \mathcal{E_T})$ *is a set of data nodes* $\mathcal{N_M} \subseteq \mathcal{N_T}$ *governed by a one-to-one mapping:*

$$f : \mathcal{N_{PQ}} \to \mathcal{N_M}$$

such that

$$\forall N \in \mathcal{N_{PQ}},\ label(f(N)) = label(N), type(f(N)) = type(N) \quad and$$

$$\exists PQ\text{'} = (\mathcal{N_{PQ'}}, \mathcal{E_{PQ'}}) \in Extension(PQ),\ \forall (E_1, E_2) \in \mathcal{E_{PQ'}},\ (f(E_1), f(E_2)) \in \mathcal{E_T}$$

where Extension(PQ) is the extension set of PQ. The edge (E_1, E_2) *or* $(f(E_1), f(E_2))$ *represents either a parent-child or an ancestor-descendant relationship.*

A potential answer to a path query is any path of our unified data tree (see Figure 1.1 in Chapter 1) that is a match to one or more of the query conditions. Similar to many popular search approaches, we focus on a ranked query model where only the k best

[3]If a term appears multiple times in a leaf node and the data path above the leaf node matches a path query, we count each occurrence of the term as a match.

matches are returned to the user. The score of a match depends on the "closeness" of the match, as defined by a scoring function (see Section 5.2).

While our model supports all possible granularity of query result (file, group of files, path segment within a file), for simplicity our current implementation only considers individual files as potential answers. We call the lowest node in a path match a *match point*. A file is an answer if its structure (including the full pathname and internal structure) and content contain the match point of a path match.

5.2 Scoring Framework

For the previous scoring framework in Chapter 3, we only consider *IDF* scores. Content (keyword) search techniques have demonstrated the importance of *TF*. In this chapter, because we are unifying content and structure, we take the opportunity to extend our scoring framework to include a *TF* component. Our scores allow for simultaneous approximation in the structure and content (Section 5.2.1). We score the answers to individual paths of the query twig with *IDF* and *TF* scores; individual path scores are then combined together to produce a unified score (Section 5.2.2).

5.2.1 Query Relaxations

Our strategy is to compute scores for answers based on how close they match the original query conditions. For content, closeness is defined based on the number of keywords from the query condition is contained in the answer (and their frequency). For structure, we use query relaxations, i.e., structural query transformations that are based on relaxations steps. A match to a relaxed version of a structure query condition is then an approximate match, with the degree of approximation depending on the number of relaxation steps.

We extend the relaxations defined in Chapter 3 with relaxations that mix content and structure conditions and take into consideration the structure within a file to handle unified content and structure queries. Similarly, we require that answers to a path query PQ be contained in the set of answers to any relaxation of PQ to ensure monotonicity of *IDF* scores (since *IDF* scores depend on the number of files that are answers to the path query).

Besides *Edge Generalization* and *Path Extension* that are already defined in Section 3.1.4.2, we consider the following new or extended structural relaxation operations:

- **Node Generalization** is used to relax a leaf structure or content node to an generalized node. This relaxation allows structure conditions to be approximately matched by content and vice versa, and is critical to our unified approach.

  For example, applying node generalization on */home/Halloween* would result in the query */home/*{*Halloween*}, which means the term "Halloween" could be part of the external directory structure, internal file structure, or file content (represented as leaf nodes in our unified structure).

- **Node Inversion** is used to permute nodes within a path query *PQ*. Except for the root, non-generalized leaf, and * nodes, permutations can be applied to any adjacent nodes or node groups if all the surrounding edges of the node or node group are ancestor-descendant edges. A permutation combines adjacent nodes, or node groups, into a single node group while preserving the placement of the generalized node, if any.

  For example, applying node inversion on *Halloween* and *witch* from */home//Halloween //witch* results in */home//(Halloween//witch)*. Applying node inversion on the same pair of nodes from */home//Halloween//*{*witch*} would result in */home //(Halloween//*{*witch*}*)*, which allows for */home//witch//*{*Halloween*} in addition to the original query condition.

  By excluding non-generalized leaf and parent-child edges from node groups, we reduce the number of relaxed queries and thus controls the overhead of query processing. In future, we will do a more thorough study to determine the types of relaxed queries that have little impact on scores, and thus can be ignored.

- **Node Deletion** is used to drop a node from a path query. This operation is similar to Node Deletion introduced in Section 3.1.4.2, except it now needs to handle the generalized node properly.

To delete a node n in a path query PQ, we take the same steps as the ones that are described in Section 3.1.4.2.

To delete a node n that is within a node group NG in a path query PQ, the following steps are required to ensure containment and monotonicity of *IDF* scores:

- n and one of its adjacent edge in NG are dropped from PQ. If only one node m is left in NG, NG is replaced by m in PQ.
- If NG is a leaf node group, the result query is extended.[4]

These steps are simpler than the ones that are described in Section 3.1.4.2, since now all edges in NG are ancestor-descendant edges.

For example, applying node deletion on *Halloween* from *//(home//Halloween//{witch})* results in *//(home//{witch})//**. This is because the extension set of *//(home //Halloween//{witch})* contains 6 path queries, which include *//home//Halloween //{witch}*, *//witch//Halloween//{home}*, and *//home//witch//{Halloween}*; after deleting *Halloween*, these path queries become *//home//{witch}*, *//witch//{home}*, and *//home//witch//**. *//(home//{witch})//** is the only most specific path query that contains the complete extension set and does not contain *Halloween*.

Our relaxation operations can be composed to provide increasingly relaxed versions of the original path query. For any path query PQ the most general relaxation is *//**, which matches all files in the unified data tree.

5.2.2 Scoring Methodology

The content *TF·IDF* scoring strategy has been well understood and widely accepted in IR community. We extend this idea and develop our unified scoring strategy which is based on both an *IDF* component, as defined in Section 3.1.4.4, and a *TF* component.

[4]Since we allow labels to permute in NG, n could appear as a leaf node and node deletion results in an extended path query. To ensure containment condition, node deletion on NG also results in an extended query.

Unlike the previous works that consider content and structure as separate dimensions scored individually and aggregated together, the novelty of this approach is that both content and structure are scored together and our scoring framework allows for approximation within and across both dimensions.

5.2.2.1 IDF Score

We first define the *IDF* score of a path query. It is similar to the scoring formula in Section 3.1.4.4, except that matches are not limited to files. Intuitively, the *IDF* of a rarely matched query is high, whereas the *IDF* of a frequently matched query is likely to be low.

Definition 13 (IDF Score of a Path Query) *Given a unified data tree T and a path query PQ, the* IDF *score of PQ is computed as:*

$$score_{idf}(PQ) = \frac{log(\frac{N}{N_{PQ}})}{log(N)}, \quad N_{PQ} = |matches(T, PQ)| \tag{5.1}$$

where $matches(T, PQ)$ is the set of all path matches to PQ in T, and N is the total number of potential matches in T.

While N could potentially be all single nodes in the unified data tree T if we allow for answers to the query to be at different granularity levels, the current implementation only considers physical files as possible matches, and thus N is the number of files in T.

5.2.2.2 TF Score

Given an answer to a path query, we define its *TF* score based on the number of path matches in the answer. Specifically, we count the number of structure and number of content match points in an answer and then normalize both counts to a value between 0 and 1 by the number of structure and content nodes in an answer respectively to alleviate the score skewness caused by different sizes (number of nodes) of answers. Finally, the normalized *TF* counts are combined together to produce the *TF* score.

Definition 14 (TF Score of an Answer for a given Path Query) *Given a path query PQ, the* TF *score of an answer A with respect to PQ is computed as:*

$$score_{tf}(PQ, A) = f\left(\frac{A_{PQ}^{(structure)}}{|A^{(structure)}|}\right) + f\left(\frac{A_{PQ}^{(content)}}{|A^{(content)}|}\right) \qquad (5.2)$$

where $|A^{(structure)}|$ and $|A^{(content)}|$ are the number of structure and content nodes in A respectively, $A_{PQ}^{(structure)}$ and $A_{PQ}^{(content)}$ are the number of structure and content match points in A that match PQ, and $f(x)$ is a function that controls the relative impact on structure and content scores, as detailed below.

In Definition 14, function f is selected from a class of functions. These functions are widely used alternatives for *TF* from the information retrieval community. In a real system, function f is selected once and then fixed for all queries and answers to compute *TF* scores. The different choice of f changes the distribution of score values, which affects the relative impact of *TF* vs. *IDF* on the unified scores as defined in Section 5.2.2.3. We will experiment with different variants of f to find the best scoring formula for our data set in Section 5.4. Specifically, f is selected from the set of functions $\{\log(1 + x)\} \cup \{x^{\frac{1}{n}} | n = 2, 3, \ldots\}$.

By definition, the more match points are contained in an answer, the higher the *TF* score of the answer.

5.2.2.3 Unified Score

In Chapter 3, we used *IDF* scores to measure the closeness of an answer to the original query. The *IDF* score of an answer is defined by the least relaxed version of the original query it matches. Answers that match the same least relaxed query receive same *IDF* score.

Given *TF* scores, however, it is important to consider all relaxed forms of the query because it is difficult to determine the single best matching relaxed form for a given file. As the query becomes more relaxed, the *IDF* score is guaranteed to decrease (as already mentioned). However, *TF* may increase significantly because a more relaxed query may have many more match points in the file. Thus, the *TF·IDF* score of a file may be higher for a more relaxed form of the query. As shall be seen, we use the summation of *TF·IDF* scores to provide an overall picture of how closely a file matches a given query by counting matching contributions across all possible relaxed forms of the query.

We define the unified score as follows.

Definition 15 (Unified Score) *Given a unified data tree T and a path query PQ, the unified score of an answer A with respect to PQ is computed as:*

$$score(PQ, A) = \sum_{PQ' \in R(PQ)} score_{tf}(PQ', A) \cdot score_{idf}(PQ') \tag{5.3}$$

where $R(PQ)$ is the set of all possible relaxations of PQ.

Finally, we compute the score of an answer A for a twig query TQ by averaging the scores of the answer for all path query decomposition PQ of TQ.

Since the current implementation only considers physical files as possible matches, A is a file in the unified data tree T.

5.2.2.4 Vector Space Model

Interestingly, our definition of unified scores (Definition 15) can be interpreted based on a vector space model.

Given a unified data tree T, we may construct path queries using the labels in T and the rules described in Section 5.1.2. Each path query represents a possible structural relationship among the data nodes. We then define a *vector space* on top of all possible path queries, where each dimension corresponds to a path query.

We may view each document d as a *vector* $\vec{v}(d)$ with each component corresponding to a possible structure relationship (path query) in the vector space, together with a weight for each component that is given by $score_{tf}(PQ, d) \cdot score_{idf}(PQ)$ (Equation 5.3). For a structural relationship that cannot be matched by a document, i.e. the structural relationship does not exist in the document, this weight is zero. In an extreme case, no structural relationships exist between nodes and thus a document only contains trivial structural relationships such as $//N$ and $//“N”$. Our vector space model degrades to the traditional vector space model, where each dimension corresponds to a term in the dictionary of T.

We can also view a unified query q as a vector $\vec{v}(q)$ in the same vector space. The weight for each component indicates how important a particular dimension is for the query. For a structural relationship that cannot be derived from the original query through relaxation rules, this weight is zero.

If $\vec{v}(q)$ is an all-ones vector and we assign to each document d a score equal to the dot product

$$\vec{v}(q) \cdot \vec{v}(d),$$

this score is equivalent to our unified score defined in Equation 5.3.

5.2.2.5 Approximation to the Unified Ranking

As we pointed out in Chapter 4, it is not trivial to compute the *IDF* score of a path query. Clearly it is more expensive to compute and combine *IDF* and *TF* scores across all relaxations of a path query as we do in Equation 5.3. Therefore, it is desirable to develop an approximate ranking mechanism that sorts answers in an order that is a tight approximation to unified rank ordering based on Equation 5.3. Furthermore, it should be practical to compute this approximation.

As we discussed earlier, to determine a unified scoring formula, we first need to select a function f for *TF* (Equation 5.2). We experimented with different variants of f and found that the unified scoring formula (Equation 5.3) gives best ranking when f is the 10-th root function for our data set (Section 5.4.2), and thus we approximate the unified scoring with *TF* scores as follows.

$$score_{tf}(PQ, A) = \left(norm(PQ, A)^{(structure)}\right)^{\frac{1}{10}} + \left(norm(PQ, A)^{(content)}\right)^{\frac{1}{10}} \tag{5.4}$$

where $norm(PQ, A)^{(structure)} = \frac{A_{PQ}^{(structure)}}{|A^{(structure)}|} \in [0, 1]$ and $norm(PQ, A)^{(content)} = \frac{A_{PQ}^{(content)}}{|A^{(content)}|} \in [0, 1]$.

As shown in Equation 5.5, the n-th root of x can be approximated by a linear function if x is close to 1.

$$f(x) = x^{\frac{1}{n}} \approx 1 + \frac{1}{n}(x-1) = \frac{n-1}{n} + \frac{x}{n},\ \ x \in [0,1] \tag{5.5}$$

Given the range of values of *TF* scores, we consider they are close to 1. Therefore, based on Equation 5.4 and Equation 5.5, we approximate the unified score in Equation 5.6 as follows.

$$\begin{aligned} \mathit{score}(PQ, A) &= \sum_{PQ' \in R(PQ)} \mathit{score}_{tf}(PQ', A) \cdot \mathit{score}_{idf}(PQ') \\ &\approx \sum_{PQ' \in R(PQ)} \mathit{score}_{idf}(PQ') \cdot \left(\frac{9}{5} + \frac{\mathit{norm}_{s+c}(PQ', A)}{10} \right) \\ &= \frac{9}{5} \sum_{PQ' \in R(PQ)} \mathit{score}_{idf}(PQ') \cdot \left(1 + \frac{\mathit{norm}_{s+c}(PQ', A)}{18} \right) \\ &= \frac{9}{5} \left(\sum_{\substack{PQ' \in \\ R(PQ)}} \mathit{score}_{idf}(PQ') + \frac{1}{18} \sum_{\substack{PQ' \in \\ R(PQ)}} \mathit{score}_{idf}(PQ') \cdot \mathit{norm}_{s+c}(PQ', A) \right) \end{aligned} \tag{5.6}$$

where $\mathit{norm}_{s+c}(PQ, A) = \mathit{norm}(PQ, A)^{(structure)} + \mathit{norm}(PQ, A)^{(content)}$.

Since $\sum_{PQ' \in R(PQ)} \mathit{score}_{idf}(PQ')$ receives a much higher weight than the rest of Equation 5.6, the unified rank ordering can be approximated by the lexicographical ordering of $(\sum_{PQ' \in R(PQ)} \mathit{score}_{idf}(PQ'), \sum_{PQ' \in R(PQ)} \mathit{score}_{idf}(PQ') \cdot \mathit{norm}_{s+c}(PQ', A))$.

Our experiments show that the ordering based on $\sum_{PQ' \in R(PQ)} \mathit{score}_{idf}(PQ')$ and $\mathit{score}_{idf}(LRQ_A^{PQ})$ are almost identical, where LRQ_A^{PQ} is the least relaxed query, i.e., the relaxed query with the highest *idf* score, that answer A matches.[5] This hints that we can approximate the unified rank ordering by the lexicographical ordering of

$$(\mathit{score}_{idf}(LRQ_A^{PQ}), \mathit{score}_{tf}(LRQ_A^{PQ}, A). \tag{5.7}$$

This lexicographical ordering is practical to compute since we only need to compute *IDF* and *TF* scores of A with respect to a single query LRQ_A^{PQ}. In doing so, we

[5]If A matches multiple relaxed queries with the same *idf* score, we choose the first such query encounted when exploring the query relaxation space as the LRQ. We could potentially search for the query giving the maximum *tf* score but this more complex approach prevents the use of an important query processing optimization (*DAGJump* algorithm in Section 4.2.2) without significantly affecting our experimental results.

save the query processing time that is spent for all relaxed queries of PQ, required by Equation 5.3, except LRQ_A^{PQ}. In Section 5.4, we will provide experimental evidence that Equation 5.7 based lexicographical ordering is a tight approximation to the unified rank ordering for our data set.

5.3 Query Evaluation

We now describe our query evaluation techniques, based on a top-k or ranked query model, which return the best k answers to the users. A benefit of such a model is that not all potential query matches need to be evaluated but only the most promising ones. We start by detailing our main index structures in Section 5.3.1. We present our algorithms to evaluate our flexible queries over our unified data set in Section 5.3.2. Finally, we discuss our main top-k query evaluation in Section 5.3.3.

5.3.1 Index Structures

We index data from our unified data tree (Section 5.1.1) using persistent inverted lists structures similar to those used in the XML community [92]. In contrast, because the possible query relaxations (Section 5.2.1) are query dependent, we need to build structures to evaluate them at runtime.

5.3.1.1 Indexing Data

We use an inverted index to enable fast access to our (potentially very large) unified data tree. Each node of the tree is assigned an identifier of the form *(FileId, PreCode:PostCode, Depth)*, where (a) *FileId* is the identifier of the file containing N if N is part of a file subtree, or 0 if N is a directory; (b) *PreCode* and *PostCode* are the values generated by a preorder and postorder traversal of the tree respectively; and (c) *Depth* is the number of edges on the path from *root* to N in the data tree.

The *PreCode:PostCode* and *Depth* information are used to quickly determine the structural relationships (ancestor-descendant and parent-child) and are widely used in XML query processing [52]. The *FileId* information is used in our implementation to quickly identify answers (files) that match a particular query.

Labels are extracted from structure and content nodes in the unified data tree. Each unique label is an entry in the inverted index.

This inverted index differs from the one that is used in Section 4.1.3 in that it uses *PreCode:PostCode* instead of *DirID* to encode the positional information of data nodes. The index containing *PreCode:PostCode* is more compact (the other index duplicates the same *DirID* for each term in the full pathname), while it has bigger overhead for data updates when the data tree changes (the order between *PreCode*s and *PostCode*s has to be maintained). We use *PreCode:PostCode* for the inverted index since the unified data tree contains far more data nodes than the directory structure.

5.3.1.2 Indexing Query Relaxations

We represent all possible relaxations of a query, along with the corresponding *IDF* scores for (files that match) each relaxation, using a DAG structure, as was proposed in Section 3.1.4.3. The DAG is created by incrementally applying query relaxations to the original query condition. Children nodes of a DAG node are more relaxed versions of the query and therefore match at least as many answers as their parent (containment property). The *IDF* score associated with a DAG node can be no greater than the score associated with its ancestors. As we go down the DAG, the increasingly relaxed versions of the query are matched by more and more files resulting in lower *IDF* scores. The most relaxed version of the query: *//** matches all files and has a score of 0.

In Section 4.2 we have detailed algorithms to efficiently build such a DAG structure. We adapt these algorithms to handle our full set of relaxations described in Section 5.2.1.

5.3.2 Query Pattern Matching

We now present our query pattern matching algorithm (Section 5.3.2.1) and analyze its time and space complexities (Section 5.3.2.2).

5.3.2.1 NIPathStack Algorithm

Given a query, we need algorithms to efficiently use the indexing structures just described to identify and score matching answers. Several efficient algorithms have been proposed for XML pattern matching, one of the most popular being the *PathStack* algorithm [18]. *PathStack* views the XML data tree as a stream of nodes produced by a preorder traversal. The algorithm associates a stack S_N with each query node N, keeping the stack in the same order as the query nodes. It then pushes matching data nodes from the stream onto the stacks. Whenever a node is pushed onto the last stack, each unique sequence of nodes across all the stacks, one per stack, that satisfies the structural relationships in the query is an answer to the query. Nodes are popped from the stacks when processing moves to a different tree branch.

Since our query model includes node permutations in the form of node groups (Section 5.1.2), *PathStack*, which imposes a strict order between query nodes, cannot be directly applied to our query. Node group permutations are costly to evaluate as the size of the potential matches grows exponentially. We propose the following *PathStack* adaptation to perform efficient query pattern matching for path queries that include node inversions. Our method contains two steps:

1. For each node group NG in a query PQ we use our new *NIPathStack* algorithm to find all matches in the unified data tree and return them in increasing order of $PreCode$. A match for a node group typically contains multiple data nodes. We first sort the matches to a node group by $PreCode_t$ and then by $PreCode_h$ in case of ties, where $PreCode_t$ and $PreCode_h$ are the $PreCode$ of the tail and head node of the match. In fact, $PreCode_t$ and $PreCode_h$ are the lowest and highest $PreCode$ values for the nodes of a match.

2. We apply *PathStack* on PQ with a small variation: each node group NG is given an individual stack, and its matches are populated by the result of *NIPathStack*. Non-node group nodes are given a stack each as in the original *PathStack*.

Like *PathStack*, *NIPathStack* uses a set of stacks, one per query node in a given node group, to find matches in our data tree and views the data tree as a stream of nodes

Algorithm 6 *NIPathStack(NG)*

```
1.  $Tail \Leftarrow nil$
2.  while $\neg end(NG)$ do
3.    $N_{min} \Leftarrow getMinSource(NG)$
4.    while $Tail \neq nil \wedge postCode(Tail) < nextPre(T_{N_{min}})$ do
5.      $pop(S_{Tail})$    {$S_{Tail}$ is the stack containing node $Tail$.}
6.      $Tail \Leftarrow prevDataPathNode(Tail)$
7.    $moveNodeToStack(T_{N_{min}}, S_{N_{min}}$, pointer to $Tail)$)
8.    $Tail \Leftarrow top(S_{N_{min}})$
9.    if $containSolution(NG)$ then
10.     $NIShowSolutions(Tail, NG, S)$
11. function $end(NG)$
12. begin
13.   return $\forall N_i \in NG : eof(T_{N_i})$
14. function $containSolution(NG)$
15. begin
16.   return $\forall N_i \in NG : |S_{N_i}| \geq 1$
17. function $getMinSource(NG)$
18. begin
19.   return $N_i \in NG$ such that $nextPre(T_{N_i})$ is minimal
20. function $moveNodeToStack(T_N, S_N, p)$
21. begin
22.   $push(S_N, (next(T_N), p))$
23.   $advance(T_N)$
```

produced by a preorder traversal. However, we use our inverted index (Section 5.3.1) to avoid traversing the entire tree. Specifically, each query node is associated with the inverted list keyed by the node's label or value. We then simulate the preorder traversal by considering the nodes from the matching inverted lists in sorted order according to their *PreCode*. As nodes are pushed onto the stacks, NIPathStack maintain pointers between these nodes to represent structural ancestor-descendant relationships in the data. As each new node is pushed onto one of the stack, the algorithm checks for solutions. At least one solution exists if all stacks are populated since node groups allow for any ordering of nodes within answers. When the traversal passes the leaf data node of a branch, nodes that cannot be involved in any new matches are popped from the stacks and the traversal moves to the next branch in the unified data tree.

Algorithm 6 details the *NIPathStack* algorithm. *NIPathStack* takes as input the node group being evaluated *NG*. It keeps a pointer *Tail* to track the deepest node of the data path encoded in the stacks. Line 3 identifies the stream to be processed

Algorithm 7 $NIShowSolutions(Tail, NG)$

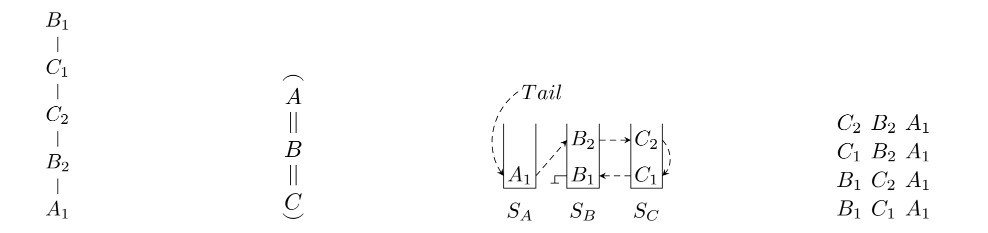
```
1.  $P \Leftarrow (Tail)$ {partial answer with one node}
2.  $showSolution(prevDataPathNode(Tail), P, NG)$

3.  function $showSolution(n, P, NG)$
4.  begin
5.    if $P$ does not contain a node with same label as $n$ then
6.      $P \Leftarrow (P, n)$
7.      if $|P| = |NG|$ then
8.        $output(P)$
9.      else
10.       $showSolution(prevDataPathNode(n), P, NG)$
11.     $P \Leftarrow P - n$
12.     if $S_n$ has at least one node below $n$ then
13.       $showSolution(prevDataPathNode(n), P, NG)$
14.   else
15.     if $|P| < |NG| \wedge prevDataPathNode(n) \neq nil$ then
16.       $showSolution(prevDataPathNode(n), P, NG)$
```

to simulate the preorder traversal, i.e., the stream which contains the node with the minimum PreCode. The function *nextPre* returns the PreCode of the next node in a stream. Line 4-6 pop stack nodes if the algorithm moves to the next branch during a preorder traversal of the unified data tree. The function *prevDataPathNode* is used to return the previous node in the data path encoded in stacks. Line 7-8 augment the data path with the new data node and assign the new deepest node of the data path. If at least one solution exists (function *containSolution* checks if all stacks are populated), line 10 invokes Algorithm 7.

We use Algorithm 7 to output all answers that ends with node *Tail*. The answers are composed recursively in leaf-to-root order. Line 1 creates a partial answer P that only contains *Tail*. Line 2 calls function *showSolution* recursively to output complete answers. Each time when *showSolution* is invoked with an input node n, it tries to output answers containing n (Line 10) and answers not containing n (Line 13 and 16) sequentially.

Figure 5.2c shows the stack encoding of the data path $B_1/C_1/C_2/B_2/A_1$ (Figure 5.2a) for the node group *(A//B//C)* (Figure 5.2b). In this example, all data nodes are part of the same path and are therefore linked together. *NISHowSolutions* outputs answers in a leaf-to-root order, starting with the deep most node (here A_1), all answers ending with this node are then produced recursively, in our example: $B_1/C_1/A_1$,



(a) Data (b) Node group (c) Stack encoding (d) Path matches

Figure 5.2: Data path and answer encoding in stacks.

$B_1/C_2/A_1$. $C_1/B_2/A_1$, and $C_2/B_2/A_1$. Because the node group semantic allows for any ordering of nodes in data paths, the algorithm is guaranteed to return at least one path match if each stack contains at least one data node.[6]

5.3.2.2 Analysis of NIPathStack Algorithm

We now prove the correctness and analyze the time and space complexity of *NIPathStack* algorithm. The analysis is similar to [18].

Lemma 3 *Suppose that t_N is an arbitrary node of the data path encoded in stacks and we have that $getMinSource(n) = N'$. And suppose that $t_{N'}$ is the next element in N''s stream. Then, after $t_{N'}$ is pushed onto stack $S_{N'}$, the data nodes from t_N to $t_{N'}$ in stacks verifies that their labels are included in a path in the unified data tree from $t_{N'}$ to the root.*

For each node $t_{N_{min}}$ pushed onto stack $S_{N_{min}}$, it is easy to see that Lemma 3 holds given the Proposition 3.1 in [18].

Lemma 4 *The the data path encoded in stacks contains at least one answer to the node group pattern NG iff $\forall N_i \in NG : |S_{N_i}| \geq 1$.*

It is also easy to see that for any query node N_i in *NG*, if $|S_{N_i}| \geq 1$, at least one data node in the chain matches N_i. Because *NG* only contains ancestor-descendant edges

[6] For simplicity, we assume node groups contain unique labels. With minor changes our algorithm can properly deal with duplicated labels in node groups, either by duplicating stacks or by adding a counter condition to stacks.

and we allow query nodes to permute within *NG*, according to Lemma 3, the data path in stacks at least contains one path that matches *NG*.

Note that the iterative nature of Algorithm 7 ensures that all possible answers that end with $t_{N_{min}}$ will be output. This leads to the correctness of our algorithms.

Theorem 16 *Given a node group pattern NG and a unified data tree T, Algorithm NIPathStack correctly returns all answers to NG in T.*

Given a node group *NG*, *NIPathStack* takes $|NG|$ input lists of data nodes sorted by *PreCode*, and computes an output sorted list of paths that match *NG*. There is one positional coding and a pointer for each element in the stacks. It is straightforward to see that, excluding the calls to *NIShowSolutions*, the I/O and CPU cost of *NIPathStack* are linear in the sum of sizes of $|NG|$ input lists.

Note that *NIShowSolutions* is called only if the data path in stacks contains at least one answer. And unless an answer can be constructed from the rest of the data path, line 8, 15, and 18 in Algorithm 7 prevent recursive invocation of *showNext*. Therefore, the cost to output each answer is bounded by the maximum length of a root-to-leaf path in the unified data tree and the cost of *NIShowSolutions* is proportional to the size of the output list.

Theorem 17 *Given a node group NG and a unified data tree T, Algorithm NIPath-Stack has worst-case I/O and CPU time complexities linear in the sum of sizes of the $|NG|$ input lists and the output list. And the worst-case space complexity of Algorithm NIPathStack is the minimum of (a) the sum of sizes of the $|NG|$ input lists, and (b) the maximum length of a root-to-leaf path in T.*

5.3.3 Top-k query Processing

Our query model returns the k best matches to a user query. Many top-k query algorithm for various query scenarios have been proposed in the literature. We have adapted the Threshold Algorithm (TA) [43] to our scenario. *TA* uses a threshold condition to avoid evaluating all possible matches to a query, focusing instead on identifying the k best answers.

TA takes as input several sorted lists, each containing the system's objects (files) sorted in descending order according to their relevance scores for a particular attribute, and dynamically accesses the sorted lists until the threshold condition is met to find the k best answers without evaluating all possible matches to a query. In our model, each list represents the answers to one of the path query decomposition of the twig query TQ. By traversing each path query DAG to access its matches in increasing degree of relaxations, we can produce a sorted list of matches for each path query based on the approximated unified scores (Section 5.2.2.5). This is ensured by the monotonicity property of the DAG IDF-based structure scores and lexicographical ordering of (IDF, TF) tuples.

In the case where the query TQ decomposes in a single path query, a simple DAG traversal, with query pattern matching along the (more and more) relaxed versions of the query will yield the answers to TQ sorted by scores. We can easily adapt *DAGJump* algorithm in Section 4.2.2 to support this sorted access efficiently, since it only requires the containment condition in a DAG. If TQ decomposes to more than one path query, then each individual path query result is seen as a sorted list on which *TA* is applied. In addition to sorted access to the list, which is provided by a top-down DAG traversal, *TA* requires random access to individual scores. Our *RandomDAG* algorithm in Section 4.2.3 can also be adapted to the unified search, as long as we can identify the query labels that are contained in a document. We simply scan the inverted index for the labels in a query once to get this information.

5.4 Experimental Evaluation

We now present the experimental results for our unified search approach.

5.4.1 Experimental Setup

Data set.

As mentioned in Section 3.2.1.2, our data set contains files and directories from our working environment. Unlike the data set in Section 3.2.1.2, here we use a new data set that is three times bigger (95,172 files in 7,788 directories) and consists mostly of document (59%) and email messages (34%). The more structure and content information contained in the new data set allows for us to explore the characteristics of our unified approach more extensively.

For this data set, the average directory depth was 6.3 with the longest being 12. On average, directories contained 13 sub-directories and files, with the largest–a folder containing unsorted backup emails–containing 5,785. The system extracted close to 700K unique stemmed content terms and close to 3K unique directory path terms. The unified (directory and file) data tree contained $\sim$57M nodes, of which $\sim$49M (86%) were leaf content nodes.

Query set.

We manually constructed 28 queries for 3 search scenarios for our case studies of unified search.

To provide a more comprehensive evaluation of unified search, we also automatically constructed a set of synthetic queries. Unlike the synthetic queries described in Section 4.3.1, our new queries are twig queries that combine the structure and content information in one query condition.

Additionally, these queries are constructed for a larger set (80) of search scenarios. Each search scenario targeted a particular file, with 20 scenarios targeting randomly chosen files from each of four different data categories, including email, document (ebooks, academic papers, etc.), music, and picture files. Queries were constructed to contain varying numbers of query conditions, as well as different combinations of structure and content terms.

More specifically, each query targeted a specific file f from the 80 search scenarios. Each query comprised n terms, where n was randomly chosen from $\{4, 5, 6\}$. The n terms were randomly selected from terms in f's directory pathname (external structure), structure terms inside f (internal structure), and f's content. To ensure reasonable selectiveness of terms, we exclude content terms that appear in more than 5,000 files. The term selection process was designed to select approximate $n/2$ external structure terms, $n/4$ internal structure terms, and $n/4$ content terms. Because some target files did not contain any internal structure, this process led to an average of 4.9 terms in each query, with 2.3 external structure terms, 1 internal structure term, and 1.6 content terms.[7]

We then construct specific queries for the different search techniques as follows:

Unified Each query is a twig with all n terms arranged according to their original positions in the unified data tree. For example, if a and c were chosen from the target file f's directory pathname */a/b/c/f* and *"foo"* from its content, then the resulting query would be *//a//c//"foo"*.

Content-only Each query contains the subset of terms selected from f's content.

Content+Dir Each query contains all n terms.

Content:Dir Each query contains all n terms but the terms are separated into two query conditions. The first contains terms selected from inside f, including both internal structure and content terms, while the second contains terms selected from f's directory pathname.

Experimental platform.

Our platform is same as the one that has been used in Chapter 3.

Relevance comparison.

We use the Lucene text search engine [47] as a comparison basis. Specifically, we compare our approach against three different approaches: content-only and two variations of content and directory path terms. For *content-only*, we use the standard Lucene content indexing and search. For the first variation of content and directory path terms (*content:dir*), we create two Lucene indexes, one of content terms and one of terms from the directory pathnames; effectively, the latter treats each directory pathname as a file with the terms (components) in the pathname being its content. Then, each query can contain two conditions, one for content and one for directory

[7]We also constructed 2 additional query sets, where content was emphasized more heavily, containing 2.5 and 3.7 content terms out of 4.9 total query terms, respectively. Further, each of these mostly contained structure information of just one type (either external or internal). The results for these query sets show similar trends as for the original query set so we do not discuss them further here.

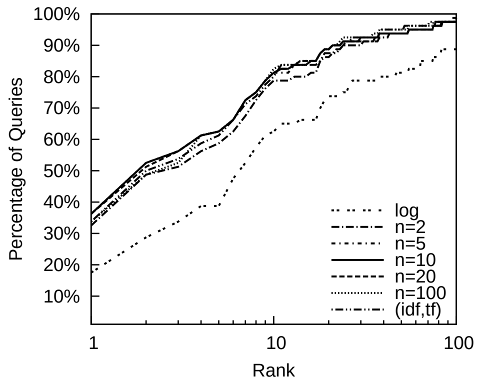
path terms. Each query condition is scored individually against the appropriate index using Lucene. The scores are then combined using a vector projection approach as described in Chapter 3. For the second variation (*content+dir*), we create a combined index that contains all content terms as well as directory path terms; terms in the pathname of each file is added to its content. Queries then contain terms that may match content or directory path terms. Queries are executed as searches against the combined index using Lucene.

We compare our unified approach to content:dir and content+dir because the latter two are plausible approaches that use some structure information (i.e., terms extracted from directory pathnames and internal structure) but are simpler to implement. Collectively, we refer to content-only, content:dir, and content+dir as "bag-of-terms" approaches because they do not consider structural relationships. We do not compare unified search against filtering approaches because the work in Chapter 3 has already shown that a flexible approach can find and rank relevant files that are missed entirely when filtering.

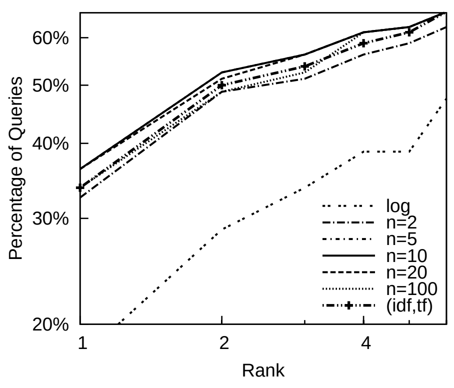
5.4.2 Approximation to the Unified Ranking

We first experimentally identify the best unified scoring formula for our *IDF* and *TF* definitions. To this end, we used the *Unified* query set described in Section 5.4.1. We brute-forcefully computed *IDF* and *TF* scores for answers to all the queries contained in each relaxation DAG associated with the query set. The scores were combined across all (relaxed) queries to produce unified scores as detailed in Section 5.2.2.3. We ranked the documents based on their unified scores.

Figure 5.3 plots the cumulative distribution function (CDF) of the rank of target files for 80 queries, where each data point on the curve corresponds to the percentage of queries (Y-axis) with the corresponding target files positioned at or higher than a particular rank (X-axis). When there are ties, we use the median value of the range as the rank of the target file; e.g., 5 files, including the target file, achieves the same highest relevance score would lead to a rank of 3 for the target file. Figure 5.3 presents CDFs for seven different variations, six where the *TF* component of unified scores were



(a)



(b)

Figure 5.3: (a) CDFs of ranks of target files for six different TF formulas ($f(x) \in \{log(1+x)\} \cup \{x^{\frac{1}{n}} | n = 2, 5, 10, 20, 100\}$) and the approximation to unified ranking based on the lexicographical (*idf,tf*) ordering. (b) An enlarged region in plot a. The curves $n = 5$ and $n = 10$ almost overlap with each other, although $n = 10$ gives slightly better ranking.

computed using either logarithm or n-th root functions, and one where documents were sorted based on lexicographical (*idf,tf*) ordering with respect to the least relaxed query that a document matches.

We observed that the logarithm function gives the worst result among the first six functions. For n-th root functions, as n increases, the ranking gradually improved and then worsened, with 10-th root function giving the best ranking for our data set. This implies $n = 10$ is optimal given our TF scoring formula (Equation 5.2).

While studing the rankings based on n-th root functions, we noticed that the rankings are not very sensitive to n. Overall the CDF varies about 5% for n between 2 and 100 and CDF curves $n = 5, 10, 20$ almost overlap each other. The unified scoring formula is stable with respect to the variations of n.

Figure 5.3 also shows that the lexicographical (*idf,tf*) ordering based on least relaxed query that an answer matches is a tight approximation to unified ranking. For any ranking position, the difference between CDF curves (*idf,tf*) and $n = 10$ is less than 3%.

Since our approximation to unified ranking is tight and it is practical to compute by only evaluating the most specific query that a file matches, we use this approximation approach throughout our implementation. For the rest of this chapter, we assume

Relative Standard Deviations				
	All Documents		**Top 100 Documents**	
Functions	*IDF*	*TF*	*IDF*	*TF*
log	0.6095	0.6806	0.3775	0.5356
$n = 2$	0.6095	0.4533	0.3663	0.4076
$n = 5$	0.6095	0.2727	0.3587	0.2814
$n = 10$	0.6095	0.2020	0.3575	0.2380
$n = 50$	0.6095	0.1690	0.3571	0.2180
$n = 100$	0.6095	0.1524	0.3564	0.2123

Table 5.1: Relative standard deviations of *IDF* and *TF* scores for six functions ($f(x) \in \{log(1+x)\} \cup \{x^{\frac{1}{n}} | n = 2, 5, 10, 20, 100\}$). Data is collected for two cases: all documents in data set and top 100 ranked documents.

(without explicit mentioning) the approximation to unified ranking is always used for comparison between unified search and other search approaches.

We now experimentally show that we have adequately considered different combinations of *IDF* and *TF* scores with respect to their impact on unified scores.

We first develop a measurement for the impact of *IDF* and *TF* scores on unified ranking, i.e. how easily their variations may change unified ordering. Since unified scores are computed as the sum of multiplication of *IDF* and *TF* scores, we are particularly interested in the sensitivity of $tf \cdot idf$ with respect to tf and idf. We observed that often bigger values (for either tf or idf) require bigger variations to attain the similar level of impact as smaller values. For example, document d_1 ($idf_1 = 0.8$, $tf_1 = 0.001$) and document d_2 ($idf_2 = 0.5 < idf_1$, $tf_2 = 0.003 > tf_2$) have different ordering with respect to idf and tf values. Even if the absolute value of difference between idf (0.3) is bigger than tf (0.002), the $tf \cdot idf$ ordering follows the ordering of tf instead of idf, since $tf_1 \cdot idf_1 = 0.0008 < tf_2 \cdot idf_2 = 0.0015$, meaning the impact of tf is bigger. If we divide their standard deviations by arithmetic means, noted *relative standard deviation* (RSD), assuming this example only involves two documents, $RSD_{idf} = \frac{0.15}{0.65} = 0.23 < RSD_{tf} = \frac{0.001}{0.002} = 0.5$ correctly reflects tf has bigger impact on $tf \cdot idf$ ordering.

We measure the impact of tf and idf by their relative standard deviations. That is, the bigger the relative standard deviation, the bigger the impact on the unified ordering.

Table 5.1 shows relative standard deviations of *IDF* and *TF* for six functions over the *Unified* query set. We only modify *TF* scoring to explore different impact of *TF* relative to *IDF*. Clearly *TF* has the smallest impact for logarithm function. As n increases for n-th root functions, *TF* attains bigger impact. When the whole document set is considered, *IDF* has bigger impact for five functions. The relative standard deviations of *IDF* are same since *IDF* scoring formula is fixed. If top 100 documents are considered, the relative impact of *IDF* drops and it has bigger impact for four functions. There are small variations in relative standard deviations of *IDF* since the set of top 100 documents varies for different unified scoring formulas.

As we can see, the unified scoring does not give best ranking for our data set when the impact of *TF* relative to *IDF* is either bigger or very small. The best rank ordering is primarily based on *IDF* ordering while it allows for *TF* to occasionally change the order of some answers.

For comparison, we compute the relative standard deviations of *TF* and *IDF* scores for Lucene. Our results show that the impact of *TF* is bigger than *IDF* when the whole document set is considered; however, the impact of *TF* is smaller than *IDF* when top 100 documents are considered, same as the best formula of our unified approach.

Based on the above results, we conclude that for our data set, our approximation to the unified ranking is tight and practical and we have adequately considered different combinations of *IDF* and *TF* scores when defining the unified score.

5.4.3 Relevance Comparison

We now compare the relevance of unified search with "bag-of-terms" approaches.

5.4.3.1 Case Studies

We start by comparing unified search with different search techniques based on several case studies.

Query Number	Query Type	Query Conditions	Comment	Rank
Search Scenario 1: The user searches for a picture of Alice wearing a witch costume taken at home on Halloween. **Target file:** /Desktop/Pictures/Disk 3/2008/Home/20081101/IMG_1391.gif (tagged with “witch” and “halloween”)				
Q1	U	//home[.//“witch” and .//“halloween”]	Accurate query conditions	1
Q2	U	//home/alice[.//“witch” and .//“halloween”]	Extraneous structure condition	1
Q3	U	//halloween/witch/“home”	Structure and content terms switched	1
Q4	C	{witch, halloween}	Accurate query conditions	20
Q5	C	{home, witch, halloween}	Structure term used as content	31
Q6	C:D	{witch, halloween} : {home}	Accurate query conditions	1
Q7	C:D	{witch, home} : {halloween}	Structure and content terms switched	245-252
Q8	C+D	{home, witch, halloween}	Accurate query conditions	20
Search Scenario 2: The user searches for the chapter “Of the Travelling of the Utopians” in the electronic book “Utopia”. **Target file:** /Laptop/eBooks/Non-Fiction/Philosophy/Utopia/OPS/main6.xml				
Q9	U	//philosophy[.//“utopia” and .//“travel”]	Accurate query conditions	1
Q10	U	//philosophy[.//“utopia” and .//chapter/“travel”]	Extraneous structure condition	1
Q11	U	//title[.//philosophy/“utopia” and .//“travel”]	Out-of-order structure conditions	1
Q12	U	//utopia/travel/“philosophy”	Structure and content terms switched	2
Q13	C	{utopia, travel}	Accurate query conditions	18
Q14	C	{philosophy, utopia, travel}	Structure term used as content	9
Q15	C:D	{utopia, travel} : {philosophy}	Accurate query conditions	4
Q16	C:D	{philosophy, utopia} : {travel}	Structure and content terms switched	291

Q17	C+D	{philosophy, utopia, travel}	Accurate query conditions	24
Search Scenario 3: The user searches for an email with the subject “Spring 2006 Tuition Payment ...”. **Target file:** /Laptop/email/local/Backup/2005_Mail_Backup/Inbox/324.xml				
Q18	U	//email[.//subject/“spring” and .//“bill”]	Accurate query conditions	3
Q19	U	//email/department[.//subject/“spring” and .//“bill”]	Extraneous structure condition	3
Q20	U	//email[.//subject/“bill” and .//“spring”]	Out-of-order structure conditions	19
Q21	U	//email[.//spring/“subject” and .//“bill”]	Structure and content terms switched	3
Q22	C	{spring, bill}	Accurate query conditions	36
Q23	C	{email, spring, bill}	Structure term used as content	340
Q24	C:D	{spring, bill} : {email}	Accurate query conditions	70
Q25	C:D	{subject, spring, bill} : {email}	Accurate query conditions	142
Q26	C:D	{bill} : {email, spring}	Structure and content terms switched	596-635
Q27	C+D	{spring, bill, email}	Accurate query conditions	75
Q28	C+D	{subject, spring, bill, email}	Accurate query conditions	153

Table 5.2: The rank of target files computed by unified search and the three bag-of-terms approaches. U denotes unified queries, C content-only queries, C:D content:dir queries, and C+D content+dir queries. Each C:D query contains two sets of terms, {content term} : {directory path terms}. A range of values for Rank means that a number of files, including the target file, received the same relevance score. We use Potter stemming for indexing and querying so that the terms “travel”, “travelling”, and “traveling” are equivalent for search scenario 2.

Table 5.2 shows queries constructed for three different search scenarios and the resulting ranking by the four different search techniques. The target files are highlighted in Figure 1.1 to give an idea of how they are placed within the data set. The queries are meant to be representative of realistic queries composed by real users. A number of the queries contain inaccuracies representing when users may misremember information about what they are searching for.

Based on the resulting rank of the target files, we make the following observations.

A small amount of structure information can significantly improve search accuracy. In the absence of errors, U always achieve significantly better ranking for the target files than C. C:D also outperforms C for search scenarios 1 and 2. For example, U and C:D both achieve a rank of 1 for queries Q1 and Q6 compared to a rank of 20 achieved by C for query Q4. In search scenario 3, C (Q22) is better than C:D (Q25) because the structure terms *email* and *subject* do not add much differentiating power.

It is important to distinguish between structure and content. In the absence of errors, C+D is always worse than U and C:D, implying that differentiating between content and structure conditions is important. When we combine the index as in C+D, terms that may have great differentiating power in the structure dimension may become diluted because they occur frequently in files' content. For example, in our data set the directory term "home" occurs frequently inside files. U and C:D separate the two term spaces and so are able to achieve a rank of 1 for queries Q1 and Q6 compared to a rank of 20 achieved by C+D for query Q8.

Structural relationships provide additional differentiating power for improving search accuracy. In search scenario 3, U was able to leverage the relationship in the *subject/"spring"* part of the query condition to achieve a ranking of 3 for the target file (Q18), whereas the inclusion of *email* and *subject* actually caused C:D to perform worse than C. *email* did not add much differentiating power while *subject* was only effective when considered together with the term "spring".

It is important to be able to relax query conditions across the content and structure dimensions. While it is important to differentiate between content and structure terms, it is also important to be able to relax across these two dimensions. This is because the user may not always remember correctly which are content and which are structure terms. When they are forced to explicitly identify content vs. directory terms, without relaxation support between the two dimensions such as in C or C:D, a mistake can drastically affect the search results. For example, switching a content and structure term drops the rank of the target file from 1 (Q6) to 245-252 (Q7) and from 4 (Q15) to 291 (Q16). On the other hand, U's processing of queries Q3 and Q12, which contain the same errors, rank the target file 1 and 2 respectively.

5.4.3.2 Automatically Generated Queries

To complement the last section, where we compared unified search with four different search techniques, we now evaluate automatically generated queries that are described in Section 5.4.1.

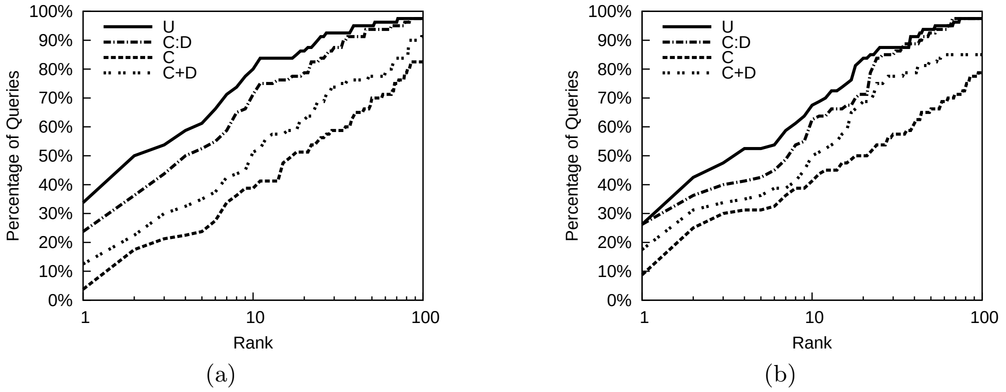


Figure 5.4: CDFs of ranks of target files for four different search techniques. In (a) content terms were selected from those close to selected internal structure terms, while in (b) content and internal structure terms were selected independently.

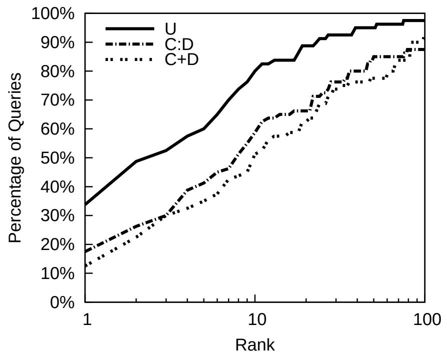
Figure 5.4 plots the cumulative distribution function (CDF) of the rank of target files for all 80 queries. CDFs are presented for two different variations, one where content terms in the queries were selected to be close to selected internal structure terms (e.g., contained in a child content node of a selected internal structure node), and one where content and internal structure terms were selected independently.

These results reinforce the first two observations that we made in the previous section: (1) A small amount of structure information can significant improve search accuracy; and (2) It is important to distinguish between structure and content; Specifically, in Figure 5.4(a), U and C:D always outperform C in ranking the target files; e.g., 80% of U queries and 71% of C:D queries ranked the target files 10 or higher, while only 39% of C queries ranked the target files within 10 or higher. Further, U and C:D queries are always better than C+D queries; e.g., only 51% of C+D queries ranked the target files 10 or higher.

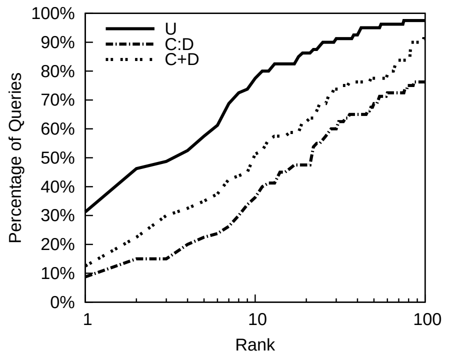
Figure 5.4 also reinforces that *unified search leverages the relationships embedded in query conditions to outperform "bag-of-terms" approaches.* In particular, Figure 5.4 shows that U outperforms C:D when internal structure and content terms are chosen close together so that relationships embedded in the query conditions are meaningful. On the other hand, their performance gets closer when content terms and internal structure terms are chosen independently, effectively having less relationships between structure and content available for U to improve its ranking accuracy.

Finally, we consider what happens if queries contain inaccuracies, where external structure and content terms are mistakenly interchanged. Figure 5.5 plots CDFs of the rank of target files when inaccuracies are introduced into the queries. We consider inaccuracies along two dimensions, the percentage of queries (chosen randomly) containing inaccurate query conditions, and the level of inaccuracy, expressed as the number of switches between pairs of external structure terms and terms from inside the target file.

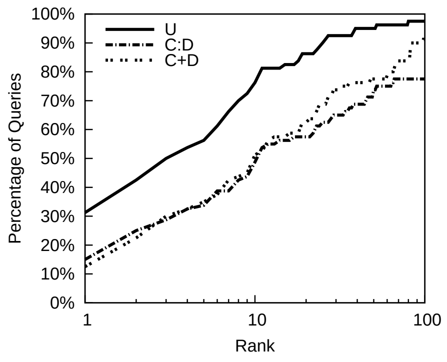
These results reinforce our final observation in Section 5.4.3.1: it is important to be able to relax conditions across the content and structure dimensions. Both U's and C:D's ranking performance degraded as the number of inaccurate queries and/or the number of inaccuracies in each inaccurate query increase. However, U queries are much less sensitive to the inaccuracies than C:D queries. In fact, Figures 5.5c and 5.5d show that C:D's can become significantly worse than C+D. On the other hand, even if 100% U queries have two pairs of directory path and content terms switched, U still outperforms C+D by a wide margin.



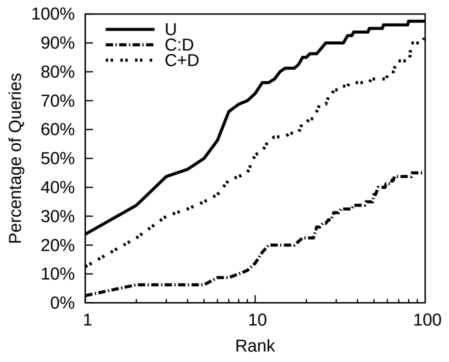
(a) 50% erroneous queries, 1 swap



(b) 100% erroneous queries, 1 swap



(c) 50% erroneous queries, 2 swap



(d) 100% erroneous queries, 2 swap

Figure 5.5: CDFs of ranks of target files when queries contain inaccuracies. For U and C:D, each figure contains a curve for 80 queries containing 50 or 100 percent erroneous queries. Each erroneous query switches one or two randomly chosen pairs of directory path and content terms. C+D is also shown as a baseline.

5.4.4 Query Processing Performance

Figure 5.6 shows the query processing times for finding and ranking the top 10 relevant files for all queries in Table 5.2. We observe that while query processing time is notably higher for unified search than all bag-of-terms approaches, it is still acceptable for everyday usage. Processing times for 7 of the 12 U queries were close to or less than 1 second. 2 of the remaining 5 took less than 3.5 seconds while 3 were around 5 seconds. For all 5, query processing time is impacted by the inclusion of a high frequency term–e.g., *title* and *subject*. Currently, such high frequency terms penalizes U heavily because they can significantly increase the time required to execute U's path pattern matching algorithm.

Figure 5.7 plots the CDF of the query processing times for the query set considered

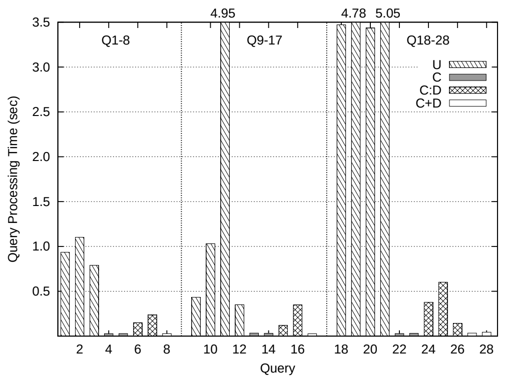


Figure 5.6: Query processing times for finding and ranking top 10 relevant files for queries in Table 5.2.

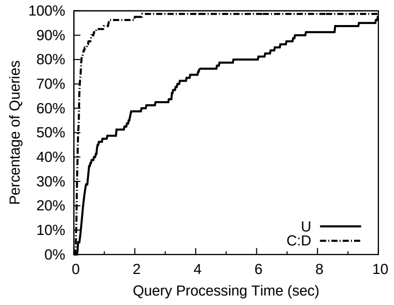


Figure 5.7: The CDF of query processing times to find and ranking top 10 relevant files for queries considered in Figure 5.4a.

in Figure 5.4a. (Query processing times are similar or better for all other query sets considered in Section 5.4.3.2 so are not shown here.) These results show that structure-heavy queries–recall that these queries on average contain 3.3 structure terms vs. 1.6 content terms–can increase query processing times. However, over 70% of the queries still require less than 4 seconds processing time, with the longest requiring 12 seconds.

5.4.5 Storage Cost

In total, our indexes require 1.9 GB of persistent storage, which is just 11% of the data set size (16.6 GB). Lucene requires 676 MB of storage to index the content of the same data set. While this is almost a three-fold increase in required persistent storage space, we believe that the total required storage is still quite reasonable. Further, it makes sense to trade-off a small amount of disk space (a plentiful resource) to improve search accuracy.

5.5 Summary

We have presented a unified framework for flexible query processing over both content and structure in personal information systems. We proposed some query processing and pattern matching algorithms to efficiently evaluate ranked search queries over our unified framework. Our experimental evaluation shows that our unified approach improves search accuracy over existing content-based methods by leveraging information from both structure and content as well as relationships between the terms. Our work shows the importance of allowing for structural query approximation in personal information queries and opens many important open research directions for efficient and high-quality search tools.

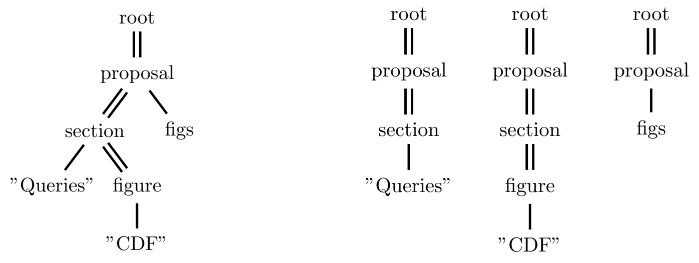
Chapter 6

Twig Queries

6.1 Introduction

In Chapter 5, we developed a scoring framework that unifies structure and content in personal information system. For simplification, we decomposed a twig query into a set of path queries for scoring. This simplified query model allows for users to provide queries containing relationships between structure and content components. Our experimental results show that relationships embedded in queries help the unified approach outperform "bag-of-terms" approaches.

Intuitively, more relationships between structure and content components may further improve search accuracy. However, the relationships embedded in twig queries are not fully utilized in the scoring framework of Chapter 5 since some of the relationship information is lost due to the twig decomposition. Specifically, without twig queries we cannot represent the relationships where multiple query nodes (components) share the same parent node.



(a) Twig query

(b) Path queries from the decomposition of twig query (a)

Figure 6.1: An example twig query and its decomposition.

For example, John, the same user as in Example 1, wants to look for *LaTeX* source and figure files (included in the *LaTeX* source files) of one of his proposals. Since logically these files constitute a single document, John wants to return them at once using a single query. Further, instead of any possible *LaTeX* source files, John is interested in a specific file that contains a section where term “Queries” and “CDF” appear in the title and a figure of the *same* section respectively. The above relationships can be represented as a twig query in Figure 6.1(a).

The methods presented in Chapter 5 would first decompose the twig query into three path queries, as shown in Figure 6.1(b). It then answers the twig query by returning files that match at least one of the path queries. Unfortunately, since searches are performed against individual path queries, we can no longer ensure the returned *LaTeX* source and figure files belong to the *same* proposal (e.g., two files are in different proposal directories that belong to different projects) and a *LaTeX* source file contains term “Queries” and “CDF” in the *same* section. To find files that satisfy these structural constraints, we would need to use twig queries.

With these observations, we believe that for personal file searches, twig queries have more representation power than path queries in at least two scenarios. First, the subtrees in twig queries can be used to represent extra constraints or the context of a search. For example, John uses two branches under “section” to ensure the term “Queries” and “CDF” appear in the same section. Second, the subtrees in twig queries can be used to represent a set of files that constitute a single logical entity. For example, John uses the subtrees rooted at “section” and “figs” to represent *LaTeX* source and figure files of the same proposal.

In depth experiments are necessary to show whether we can further improve search accuracy by using complete twig queries. However, as the first step, in this work we develop a query model for twig queries (Section 6.2) and discuss alternatives to approximate queries and score documents (Section 6.3). The solutions to problems such as flexible result granularity and efficient query matching are beyond the context of this thesis.

6.2 Data and Query Model

We now present our data and query models that support unified flexible search based on twig queries.

6.2.1 Data Model

To answer twig queries, we view the whole file system as a rooted, labeled, unordered tree, containing the internal *structure* nodes and leaf *content* nodes. This model is identical to the *unified data tree* that is defined in Section 5.1.1.

6.2.2 Query Model

Our twig query model allows users to query both content and structure of documents. Each twig query contains structure patterns and content terms such as the ones shown in Figure 5.1(a).

As we discussed in Chapter 5, users often forget details of documents and search queries are likely to be incomplete or to contain mistakes. It is desirable to allow for flexible queries to leverage any accurate structural information in queries and provide meaningful context information for content terms.

We further extend the notations in Section 5.1.2 to support this flexibility in matching against structure patterns and content terms for twig queries. Our definitions of *Root Node*, *Content Node*, *Structure Node*, *Generalized Node*, and *Extended Node* are same as the ones that are defined in Section 5.1.2. Other extended notations are defined as follows.

- **Twig Segment:** Twig segment is an extension of *Path Segment* (Section 5.1.2). They are same except that a twig segment is a rooted and unordered tree instead of a path, and it can be matched by a subtree in the unified data tree T.

- **Node Group:** We extend the definition of the node group in a path query (Section 5.2.1) by allowing permutations in twig structures. The restrictions of the node group in a path query, i.e. the types of nodes and edges that are allowed in a node group and the fixed placement of generalized node, also apply to the node group in a twig query. Additionally, each branch of a twig query contains at most one generalized node as the leaf node. A node group *NG* could contain the *root* node [1] after a node is deleted from *NG*, if the *root* node is at the parent node position of *NG* (Section 6.3.1).

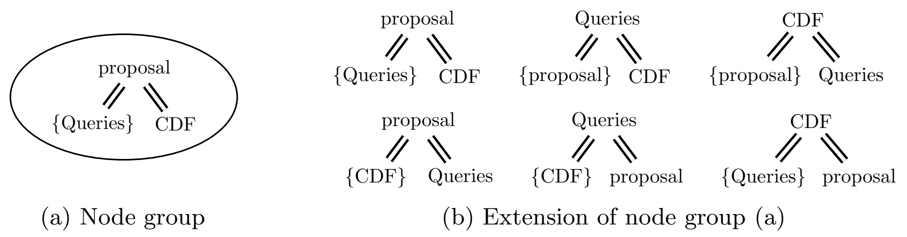


Figure 6.2: An example node group and its extension containing six twigs. The ellipse in solid line represents a node group.

Similarly to the definition in Section 5.1.2, we define the *extension* of a node group *n*, noted *Extension(n)*, as the set of all twig segments that are contained in *n*, each corresponds to a valid permutation of the nodes in *n*.

For example, Figure 6.2(a) shows a node group which corresponds to an extension set containing six twigs in Figure 6.2(b).

- **Twig Query:** We use the same definition of twig query as the one that is defined in Section 5.1.2).

A *match* of a twig query *TQ* (i.e. a *twig match*) in a data tree *T* is defined similarly to the definition of *Path Match* (Section 5.1.2). The difference is that the set of nodes of a *twig match* satisfy the embedded relationships in a tree structure instead of a path.

A potential answer to a twig query is defined as any subtree of our unified data tree that matches the twig query. We call the leaf node of each branch in a twig match a *match point*. A subtree *T'* is an answer if its structure (including the internal structure and the path between the *root* node and the root of *T'*) and content contain at least

[1]The *root* node in *NG* is fixed at the root position – the only valid position for the *root* node. This is different from the rest of nodes in *NG* since they are free to permute.

one match point of a twig match.[2] For example, to be an answer of the twig query in Figure 5.1(a), a subtree must contain the match point "Queries", "CDF", or "figs" of a twig match. This allows for *LaTeX* source and figure files to be returned at once.

In this work, we still use a ranked query model where only the best k matches are returned. The score of a match depends on the "closeness" of the match that is represented by a scoring formula (Section 6.3).

6.3 Scoring Framework

For the scoring framework in Chapter 5 (Section 5.2), we score the answers of individual path queries with *IDF* and *TF* scores. In this work, we use the same strategy and score the answers of a twig query using *IDF* and *TF* scores.

6.3.1 Query Relaxations

We first extend the query relaxations described in Chapter 5 for twig queries.

6.3.1.1 Relaxation Operations

As in Chapter 5, we require that answers of a twig query TQ be contained in the set of answers of a relaxation of TQ to ensure monotonicity of *IDF* scores when relaxing a query. Besides *Edge Generalization*, *Path Extension*, and *Node Generalization* that are already defined in Section 5.2.1, we consider the following structural relaxation operations:

- **Node Inversion** is used to permute nodes within a twig query TQ. This operation is same as the one that is described in Section 5.2.1, except that now a node group may contain branch nodes and multiple branches.

  Figure 6.3 shows two node inversion examples: one combining two nodes and one combining two node groups. The first example allows for node "proposal"

[2]We could define a subtree as an answer if its structure and content contain all match points of a twig match. We choose the current definition to allow for the maximum flexibility of result granularity since multiple answers could match a twig query collectively while none of them match the complete twig query.

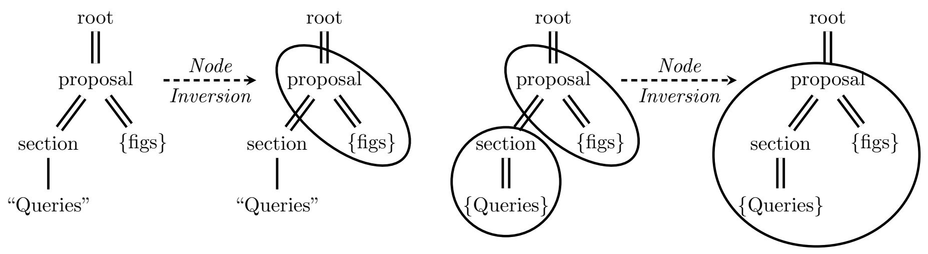


Figure 6.3: Two node inversion examples: one combining node "proposal" and "{figs}", and one combining two node groups.

and "figs" to permute and results in two twigs in the extension set of the result query. The second example allows for four nodes to permute after node inversion is applied. The result query has 24 twigs in its extension set, while the original query has two twigs in the extension set of each node group, as shown in Figure 6.4.

- **Node Deletion** is used to drop a node from a twig query. Node deletion can be applied to any query node or node group as long as their surrounding edges are ancestor-descendant edges. But it cannot be used to delete the *root* node or the *** node.

  To delete a node n in a twig query TQ, we take the same steps as the ones that are described in Section 3.1.4.2, except that it is possible *parent(n)* and multiple *child(n)* nodes are connected with // when deleting an internal node.

  If a node group NG does not contain branch nodes, to delete a node n that is within NG in a twig query TQ, we take the same steps as the ones that are described in Section 5.2.1.

  If a node group NG contains branch nodes, to delete a node n that is within NG in a twig query TQ, we cannot simply take the steps that are described in Section 5.2.1 since we have to handle branch nodes differently. If NG does not contain the *root* node, we need to add the parent node of NG to the result node group after deleting n; otherwise, we delete one node from each branch and add several new single-node branches.

  More specifically, the following steps are required to ensure answer containment

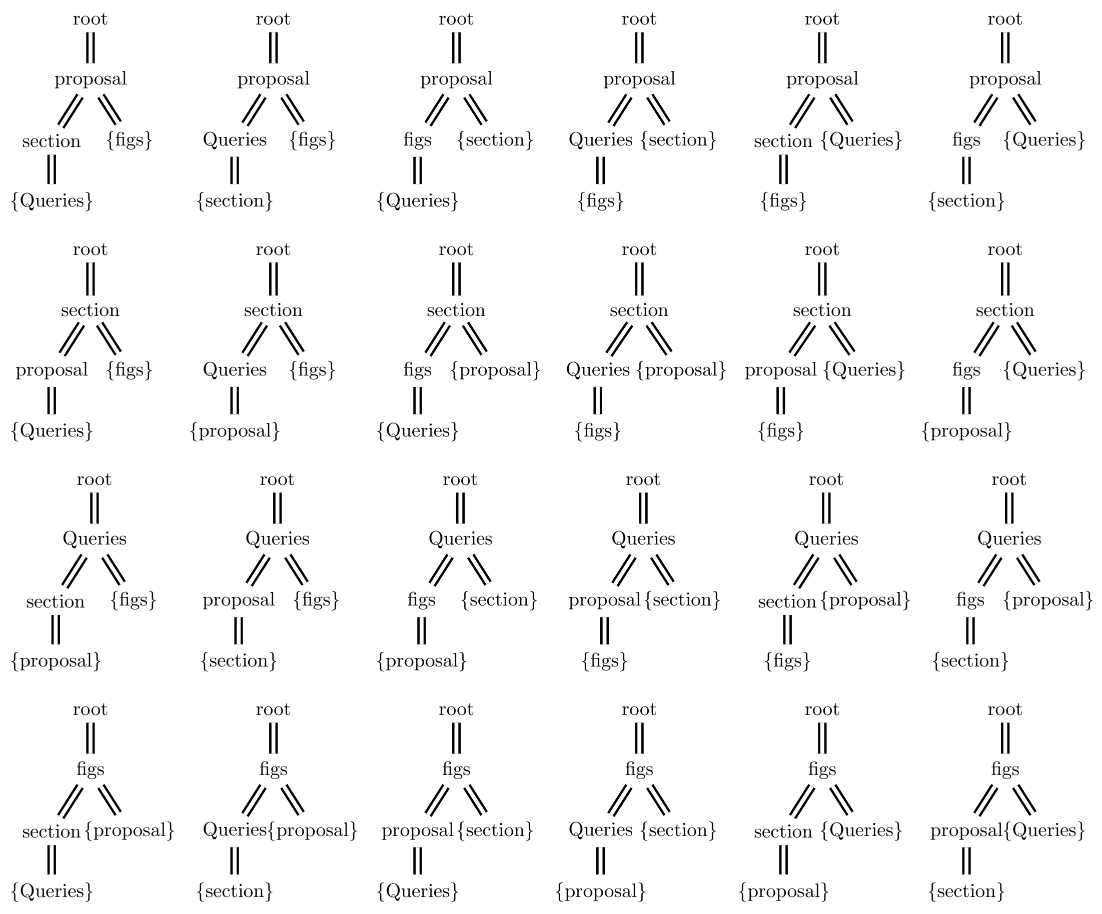


Figure 6.4: The extension set of the result query for the second node inversion example in Figure 6.3.

and monotonicity of *IDF* scores:

– If *NG* does not contain the *root* node, we apply

Rule 1.

1. The root of *NG*, noted *root(NG)*, is dropped and all subtrees of *root(NG)* are moved to the parent node of *root(NG)*, noted *parent(NG)*. All leaf nodes that are contained in *NG* are extended with *//**.[3]
2. *parent(NG)* and *NG* – *root(NG)* are combined into a new node group. All surrounding edges of *parent(NG)* are changed into //.[4]

[3]Since (labeled) nodes are free to permute in *NG*, the deleted node n may appear at any leaf node in *NG*. To ensure answer containment, the result query *TQ'* must contain all possible queries after deleting a leaf node in *NG*. Therefore, all leaf nodes of *NG* are extended with *//**.

[4]For one of the permutations of nodes of *NG*, n appears at *root(NG)*. After deleting n all branches

3. *n* is dropped from *NG*. If only one non-*root* node *N* is left in *NG*, *NG* is replaced by *N* in *TQ*.

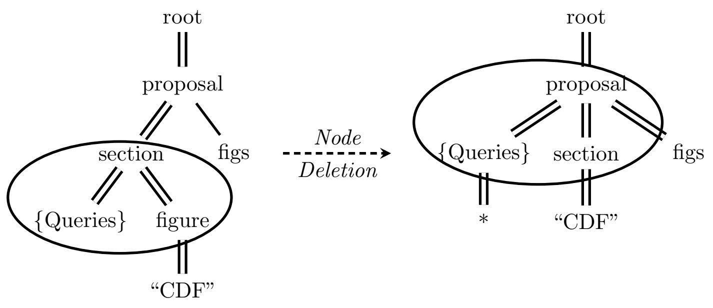


Figure 6.5: Node deletion is applied to node "figure" within a node group that does not contain the *root* node.

Figure 6.5 shows the original and result queries when node deletion is applied to node "figure" within a node group that does not contain the *root* node. The root of node group is dropped and its two subtrees are moved to its parent node "proposal". Node "proposal" is then combined into the new node group and node "Queries" is extended with //* since it is contained in the node group.

– If *NG* contains the *root* node, we apply

Rule 2.

1. For each subtree of the *root* node in *NG*, noted *ST*, we drop *root(ST)* and move all subtrees of *root(ST)* to the *root* node. All leaf nodes of *ST* that are contained in *NG* are extended with //*.[5]

2. Let $|Subtree(root)|$ be the number of subtrees of the *root* node in *NG*. $|Subtree(root)| - 1$ single-node branches are added and connected to the *root* node with //.

3. *n* is dropped from *NG*. If only one non-*root* node *N* is left in *NG*, *NG* is replaced by *N* in *TQ*.

under *root(NG)* are moved under *parent(NG)*. $NG - root(NG)$ is no longer a twig structure, and thus is not a node group unless *parent(NG)* is combined into the new node group.

[5]This is necessary for the same reason as step 1 of Rule 1.

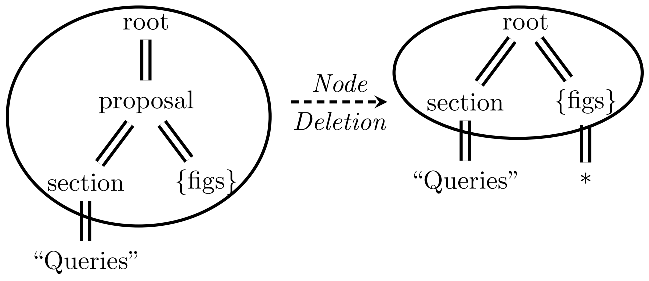
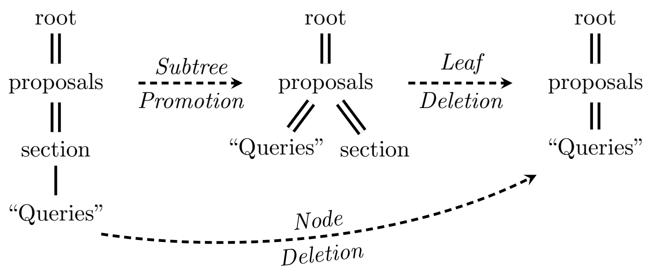


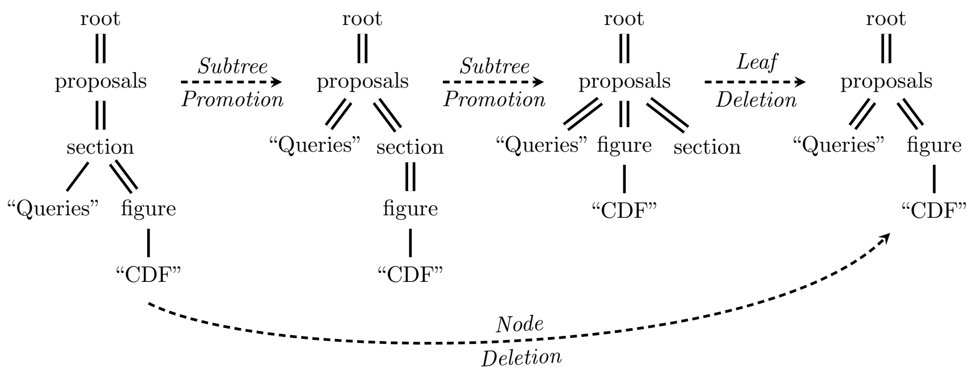
Figure 6.6: Node deletion is applied to node "proposal" within a node group containing the *root* node.

Figure 6.6 shows the original and result twig queries when node deletion is applied to node "proposal" within a node group containing the *root* node. Node "figs" is extended with //* since it is contained in the node group. The *root* node has only one subtree and no single-node branches are added.

6.3.1.2 Node Deletion vs. Subtree Promotion and Leaf Deletion



(a) Apply node deletion to node "section" within a path query.



(b) Apply node deletion to the branch node "section" within a twig query.

Figure 6.7: Example queries that show node deletion is equivalent of a series of subtree promotions followed by a leaf deletion.

Previous works [6] studied three types of XML structural relaxations: *edge generalization* (same as our definition), *leaf deletion* (making a leaf node optional), and *subtree promotion* (moving a subtrees from its parent node to its grand-parent node). To handle the specific needs of user searches in a personal information management system, we introduced new types of structural relaxations. In fact, *node deletion* is equivalent to a sequence of subtree promotions (moving a subtrees from its parent node to its grand-parent node) followed by a leaf deletion, if a twig query does not contain node groups. This is exemplified by Figure 6.7.

An alternative definition of structural relaxations is to extend the relaxation operations based on subtree promotion and leaf deletion instead of node deletion operation. A discussion of this approach is beyond the context of this thesis.

6.3.1.3 Analysis of Node Deletion

We now prove the correctness of node deletion operation. More specifically, we show that Rule 1 and 2 for node deletion satisfy the containment condition.

Correctness of Rule 1

Let *Extension(TQ)* and *Extension(TQ')* be the extension sets of the original and result queries for Rule 1 respectively. The containment condition holds if for any query $TQ_1 \in$ *Extension(TQ)*, applying node deletion to a node $n \in NG \in TQ_1$ results in a query that is less relaxed than the queries in *Extension(TQ')*.

For a rooted tree, there is one and only one path from the *root* node to another node in the tree. As shown in Figure 6.8, a twig query can be viewed as a path from the *root* node to another node in the tree, where each node on the path has a subtree along the path in addition to the rest of its subtrees (empty for a leaf node). For the discussion of Rule 1 and 3, we will focus on this path since the structural relationships in the rest of the twig query are kept intact by Rule 1 and 2.

We allow for labels to permute within a node group NG, so node n could appear anywhere in NG.

- If n is at position *root(NG)* and we delete n, the operation is equivalent to step 1

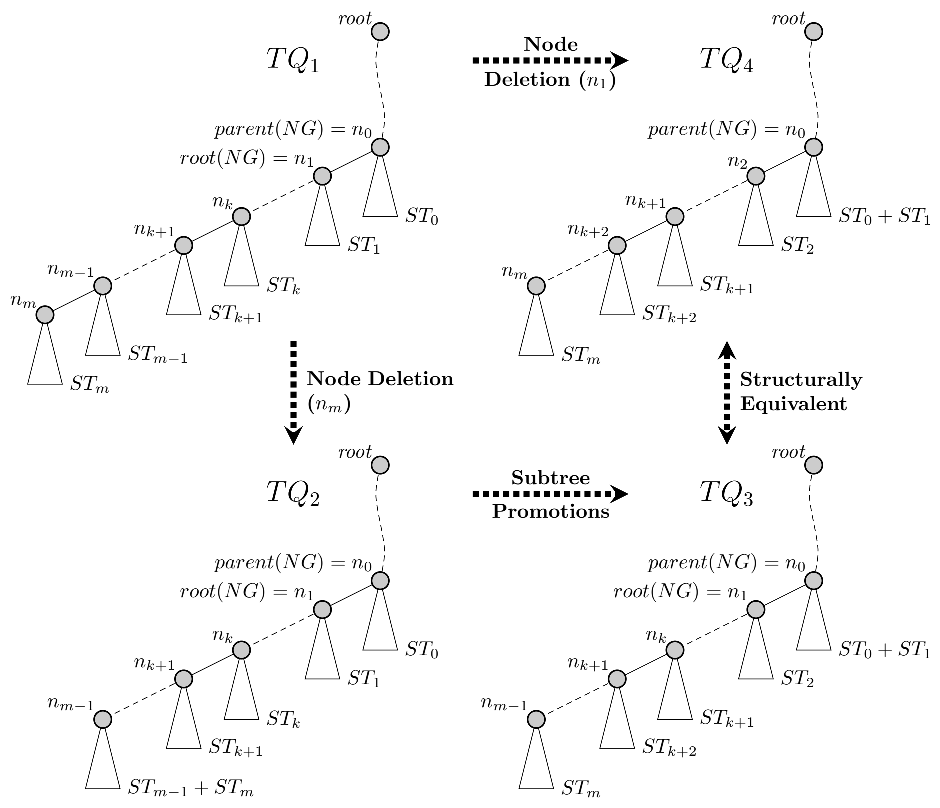


Figure 6.8: Proof of the correctness of Rule 1. Node labels represent the positions within the query. ST represents subtrees. Solid lines connect adjacent nodes. Dashed lines represent ancestor-descendant relationships, with intermediate nodes removed for simplicity of presentation.

of Rule 1. The rest of steps further relax the query. Therefore, the result query is in the extension set of the result twig query from Rule 1.

- If n is at a position n_m other than *root(NG)*, we view TQ_1 as the twig in Figure 6.8. Applying node deletion to n_m results in TQ_2. Since moving subtrees to parent nodes results in more relaxed queries (the nodes in subtrees are descendant nodes of the parent node), we further relax TQ_2 to TQ_3 using subtree promotions. TQ_3 is structurally equivalent to TQ_4 that is the result query after we apply node deletion to *root(NG)*. Since $TQ_4 \in$ *Extension(TQ')*, TQ_2 is a query that is less relaxed than the queries in *Extension(TQ')*.

Therefore, Rule 1 satisfies the containment condition.

Correctness of Rule 2

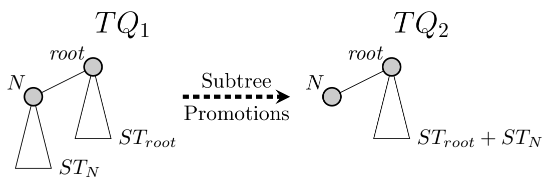


Figure 6.9: Apply subtree promotions to node N and move subtrees to the *root* node, structurally equivalent to deleting node N and add a single-node branch to the *root* node. ST represents subtrees.

Assume node group NG contains the *root* node and we want to delete a node $n \in NG \in TQ_1 \in$ *Extension(TQ)*. After deleting n, we get query TQ_2 that is less relaxed than TQ_3 (and TQ_4) in Figure 6.8, where $n_0 = root$. We then apply subtree promotions within the subtrees of *root* in TQ_3 other than the one containing node n, as shown in Figure 6.9, which is structurally equivalent to deleting node N and adding a single-node branch to the *root* node. The result query has the same structure as the queries in *Extension(TQ')* that are obtained from Rule 2. Therefore, Rule 2 satisfies the containment condition.

We need to apply node deletion for all subtrees of the *root* node within *NG* in step 1 of Rule 2. This is because we allow for nodes to permute in node group and n could appear in any subtree of the *root* node in *NG*.

6.3.2 Scoring Methodology

We now extend the unified scoring framework in Section 5.2.2 to score answers for twig queries. Similarly, the extended scoring framework is based on *IDF* and *TF* scores, and it approximates and scores both content and structure together. The differences are the new scoring framework needs to handle answers that match a twig query partially, while answers for path queries always match the complete queries.

6.3.2.1 IDF Score

The *IDF* score is defined same as the one in Section 5.2.2.1 except that the new *IDF* score are defined based on the number of twig matches instead of path matches.

6.3.2.2 TF Score

TF scores are defined based on the number (frequency) of query matches in an answer. Unlike path queries, the structural relationships of a twig query may be partially matched by an answer A. That is, A does not contain all match points of a twig match. The complete twig query is matched by a set of answers containing A. Therefore, a plausible scoring framework should consider answers that match a twig query partially and produce the *TF* score based on the portion of the twig query that is matched by an answer. One alternative is to define *TF* score based on the number of unique *match points* that are contained in an answer, i.e. we do not double count any match point even if it is contained in multiple twig matches. In doing so, the bigger portion of the twig query is matched by an answer, the bigger the *TF* score.

We define the *TF* score of an answer A to a twig query same as the one that is defined in 5.2.2.2, except that A may not necessarily contain all match points of a twig match. Specifically, we compute the *TF* score using the number of unique (structure and content) match points that are contained in A.

Similarly to Definition 14, function f could be selected from a set of functions once and then fixed for all queries and answers to compute *TF* scores. We may determine the best f function for a data set experimentally, like we did for path queries in Section 5.2.2.2.

6.3.2.3 Unified Score

We define the unified score by aggregating the *TF* and *IDF* scores across all relaxed queries that an answer matches, same as the unified score defined in Section 5.2.2.3.

In Chapter 5, we compute scores for individual path queries and then use summation of the scores of path queries to get the score of a twig query. In this chapter, we directly compute the *IDF* and *TF* scores for twig queries without twig decomposition. Different computation methods potentially could lead to different distribution of *IDF* and *TF* scores, and the lexicographical ordering of (*idf*,*tf*) scores might no longer be a tight approximation to the unified rank ordering. If (*idf*,*tf*) is not a good approximation, new experiments are necessary to determine the best formula for the approximation to unified scores.

Given the extended definition for the unified score, we can then rank the answers with their unified scores. The new challenge is to develop efficient query matching and processing techniques for twig queries so we can return k best answers of a twig query for personal information search. This is beyond the context of this thesis.

6.4 Summary

We have presented a unified framework for flexible query processing over twig queries in personal information systems. We proposed alternatives for scoring file matches of twig queries. Returning results at different granularity levels in addition to documents and efficient twig query processing are likely directions for future work.

Chapter 7

Conclusions

This dissertation investigates issues in improving searches for the often very heterogeneous data in personal information system. In particular, we focus on the algorithms and data structures that support effective and efficient search for flexible query conditions against directory structure and structure contained within files.

As users store and collect ever larger amounts of data in personal information system, there is a demand for complex search tools to retrieve the personal data in a simple and efficient manner.

Current search tools often use *ranking* on the textual part of the query, but only consider structure (e.g., file directory) and metadata (e.g., date, file type) as *filtering* conditions. However, simple keyword-only searches are often insufficient due to inflexible filtering conditions, strict separation between structure and content along file boundaries and inside files, and lack of support for queries containing relationships across multiple directories and/or files.

On the user side, while their knowledge of personal data has the potential to increase the accuracy of search, users are notoriously bad at remembering exactly where they stored a particular file or how the files are structured.

Therefore, it is desirable for search tools to be capable of (a) combining and approximating structure and content information of files, and (b) ignoring file boundaries and manipulating all personal information in a unified fashion.

To help meet the above challenges, we developed a suite of techniques to allow search tools to effectively and efficiently consider flexible query conditions along structure and content dimensions.

Our work started by considering how one can merge flexible query conditions in the structure dimension with content, using a multi-dimensional approach.

We proposed individual *IDF*-based scoring approaches for three query dimensions: content, metadata, and structure. Specifically, we developed a methodology for constructing query relaxations to support fuzzy query conditions in each dimension. We then present a unified scoring framework to produce a single unified relevance score for each file by combining individual dimension scores.

Our evaluation showed that our scoring approach provides a meaningful distribution of scores that captures the specificity of each dimension. We also demonstrated that our multi-dimensional score aggregation technique preserves the properties of individual dimension scores and has the potential to significantly improve ranking accuracy.

Next, we designed algorithms and data structures to support efficient processing of the multi-dimensional queries. The same techniques also can be adapted to unified queries.

We presented new data structures and index construction optimizations to address challenges in query processing, especially for structural query conditions. Our techniques and data structures work in conjunction with our adaptation of existing top-k algorithms to provide an efficient implementation of our overall framework.

We evaluated our implementation by executing a large number of queries against a large, real-life personal data set. Our evaluation shows that our indexes and optimizations are necessary to make multi-dimensional searches efficient enough for practical everyday usage. We also show that our optimized query processing strategies exhibit good behavior across all dimensions, resulting in good overall query performance and good scalability.

We then proceeded to consider how to unify structure and content, such as users do not have to clearly separate the two; instead, they can specify a single query that will be matched against a data structure that unifies both structure and content.

For this purpose, we use a unified data model that ignores the traditional physical file boundaries. We propose a query model that supports approximation in both the structure and content component of the queries, and allows for structure conditions to be matched by content terms and vice versa. Further, we developed a unified scoring framework that considers relaxed query conditions on structure and content at once and provide a unified score. For simplicity, we took twig queries and decomposed them into path queries. Answers are then searched against path queries and their scores of individual path queries are aggregated to produce the final unified scores.

Our experimental evaluation shows that our unified approach improves search accuracy over existing content-based methods by leveraging information from both structure and content as well as relationships between the terms. Our work shows the importance of allowing for structural query approximation in personal information queries and opens many important open research directions for efficient and high-quality search tools.

Finally, we advanced the unified structure and content search framework and turn to the complete twig queries that encapsulate all correlations between search conditions.

We focus on extending the query model of path queries for twig queries and discussing alternatives to approximate twig queries and score personal files. We also present the extended unified structure and content scoring framework that considers relaxed twig query conditions.

An interesting question raised by this work is what is the best way to define *TF* score and combine it with *IDF* score. Other possibilities exist while we defined the *TF* score based on the number of match points that an answer contains and followed the traditional content scoring strategy to combine *TF* and *IDF* scores. For example, if the leaf node of a match is a structure node, the frequency of the match could be defined as the number of content nodes in the subtree rooted at the structure node. Additionally, we could use functions other than the multiplication, or even machine learning methods, to combine *TF* and *IDF* scores, which does not necessarily follow the traditional content scoring strategy. More generally, we are starting to address the question of how to leverage the document frequency and match frequency information for scoring to archive the best possible ranking relevancy.

A second research avenue is how we can keep the index structure up to date in an dynamic and distributed environment. This is important because it is not always possible to mount all file systems to a single device and users often work on a relative small portion of their personal data for particular tasks during a period of time. A more flexible system should have partitioned index structures, built for separated file systems or different portion of personal data, that can be updated independently and allow for query evaluation on each partition to be merged to produce the final ranked results. In fact, if we divide the unified data tree (see Chapter 5) into disjoint subtrees, we can build separate inverted lists that support searches of path queries within each subtree. Search results for each subtree can then be easily merged to produce the final ranked results. However, searches for twig queries are more challenging, since a match for a twig query may cross multiple subtrees.

Finally in this work, we have focused on physical files as a result unit. We may relax this restriction to allow for logical units of data to be returned. For example, for some file types such as *LaTeX* source, users may logically think several documents as different parts of a single document. It is also very common that users exchange emails for a topic and all these emails could constitute a single logical entity. Additionally, the query model could go beyond the physical files and logical documents and return results at different granularity levels. For example, photos taken at the same time and location could be returned as a set; or the result of a search comparing job candidates could consists only of the "Education" section of resumes. Further work is required to address issues resulting from flexible granularity of returned results.

References

[1] Sanjay Agrawal, Surajit Chaudhuri, Gautam Das, and Aristides Gionis. Automated Ranking of Database Query Results. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2003.

[2] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2002.

[3] Mehmet Altinel and Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2000.

[4] Sihem Amer-Yahia, Pat Case, Thomas Rölleke, Jayavel Shanmugasundaram, and Gerhard Weikum. Report on the DB/IR panel at SIGMOD 2005. *SIGMOD Record*, 2005.

[5] Sihem Amer-Yahia, SungRan Cho, and Divesh Srivastava. Tree Pattern Relaxation. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2002.

[6] Sihem Amer-Yahia, Nick Koudas, Amélie Marian, Divesh Srivastava, and David Toman. Structure and Content Scoring for XML. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, September 2005.

[7] Sihem Amer-Yahia, Laks V. S. Lakshmanan, and Shashank Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[8] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley, 1999.

[9] Ricardo A. Baeza-Yates and Mariano P. Consens. The Continued Saga of DB-IR Integration. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2004.

[10] Xinlong Bao, Jonathan L. Herlocker, and Thomas G. Dietterich. Fewer Clicks and Less Frustration: Reducing the Cost of Reaching the Right Folder. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*, 2006.

[11] Deborah Barreau and Bonnie A. Nardi. Finding and Reminding: File Organization from the Desktop. *ACM SIGCHI Bulletin*, 27, 1995.

[12] Ofer Bergman, Ruth Beyth-Marom, Rafi Nachmias, Noa Gradovitch, and Steve Whittaker. Improved Search Engines and Navigation Preference in Personal Information Management. *ACM Transactions on Information Systems (TOIS)*, 2008.

[13] Tristan Blanc-Brude and Dominique L. Scapin. What do People Recall about their Documents?: Implications for Desktop Search Tools. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*, 2007.

[14] Lukas Blunschi, Jens peter Dittrich, Olivier René Girard, Shant Kirakos, Karakashian Marcos, and Antonio Vaz Salles. A Dataspace Odyssey: The iMeMex Personal Dataspace Management System. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, pages 114–119, 2007.

[15] C. Mic Bowman, Chanda Dharap, Mrinal Baruah, Bill Camargo, and Sunil Potti. A File System for Information Management. In *Proceedings of the International Conference on Intelligent Information Management Systems (ISMM)*, 1994.

[16] Jan-Marco Bremer and Michael Gertz. XQuery/IR: Integrating XML Document and Data Retrieval. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2002.

[17] Nicolas Bruno, Luis Gravano, Nick Koudas, and Divesh Srivastava. Navigation- vs. Index-Based XML Multi-Query Processing. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2003.

[18] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2002.

[19] Yuhan Cai, Xin Luna Dong, Alon Halevy, Jing Michelle Liu, and Jayant Madhavan. Personal Information Management with SEMEX. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, June 2005.

[20] D. Carmel, Y. S. Maarek, M. Mandelbrod, Y. Mass, and A. Soffer. Searching XML Documents via XML Fragments. In *Proceedings of the ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, 2003.

[21] Surajit Chaudhuri, Raghu Ramakrishnan, and Gerhard Weikum. Integrating DB and IR Technologies: What is the Sound of One Hand Clapping? In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2005.

[22] Jidong Chen, Hang Guo, Wentao Wu, and Chunxin Xie. Search Your Memory! – An Associative Memory Based Desktop Search System. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2009.

[23] Yi Chen, Susan B. Davidson, and Yifeng Zheng. BLAS: An Efficient XPath Processing System. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[24] Sergey Chernov. Task Detection for Activity-based Desktop Search. In *Proceedings of the ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, 2008.

[25] Taurai Tapiwa Chinenyanga and Nicholas Kushmerick. Expressive and Efficient Ranked Queries for XML Data. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2001.

[26] Paul-Alexandru Chirita and Wolfgang Nejdl. Analyzing User Behavior to Rank Desktop Items. In *Proceedings of the International Conference on String Processing and Information Retrieval*, 2006.

[27] Andrea Civan, William Jones, Predrag Klasnja, and Harry Bruce. Better to Organize Personal Information by Folders or by Tags?: The Devil is in the Details. In *Proceedings of the American Society for Information Science and Technology*, 2008.

[28] Sara Cohen, Carmel Domshlak, and Naama Zwerdling. On Ranking Techniques for Desktop Search. *ACM Transactions on Information Systems (TOIS)*, 2008.

[29] Copernic Desktop Search. `http://www.copernic.com`.

[30] W. B. Croft, P. Krovetz, and H. Turtle. Interactive Retrieval of Complex Documents. *Information Processing and Management*, 26(5):593–613, 1990.

[31] Mary Czerwinski and Eric Horvitz. An Investigation of Memory for Daily Computing Events. In *Proceedings of the British HCI Group Annual Conference*, 2002.

[32] Mary Czerwinski, Eric Horvitz, and Susan Wilhite. A Diary Study of Task Switching and Interruptions. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, 2004.

[33] Dataspace: `http://en.wikipedia.org/wiki/Dataspace`.

[34] Claude Delobel and Marie-Christine Rousset. A Uniform Approach for Querying Large Tree-structured Data through a Mediated Schema. In *Proceedings of the International Workshop on Foundations of Models for Information Integration*, 2001.

[35] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM Transactions on Database Systems*, 28, 2003.

[36] Yanlei Diao, Peter Fischer, Michael J. Franklin, and Raymond To. Yfilter: Efficient and Scalable Filtering of XML Documents. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2002.

[37] Yanlei Diao and Michael J. Franklin. High-Performance XML Filtering: An Overview of YFilter. *IEEE Data Engineering Bulletin*, 26, 2003.

[38] Jens-Peter Dittrich and Marcos Antonio Vaz Salles. iDM: A Unified and Versatile Data Model for Personal Dataspace Management. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2006.

[39] Xin Dong and Alon Halevy. Indexing Dataspaces. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2007.

[40] Johann Eder, Alexander Krumpholz, Alexandros Biliris, and Euthimios Panagos. Self-Maintained Folder Hierarchies as Document Repositories. In *Proceedings of the Internaional Conference on Digital Libraries: Research and Practice*, 2000.

[41] Ibrahim Elsayed, Peter Brezany, and A Min Tjoa. Towards Realization of Dataspaces. *Proceedings of the International Conference on Database and Expert Systems Applications*, 2006.

[42] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Comparing Top K Lists. *SIAM Journal on Discrete Mathematics*, 17(1), 2003.

[43] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal Aggregation Algorithms for Middleware. *Journal of Computer and System Sciences*, 2003.

[44] Fellbaum. *WordNet: An Electronic Lexical Database (Language, Speech, and Communication)*. The MIT Press, 1998.

[45] Lin Guo Feng, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2003.

[46] Daniela Florescu, Inria Roquencourt, and Donald Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22, 1999.

[47] Apache Software Foundation. Lucene. `http://lucene.apache.org/`.

[48] Michael Franklin, Alon Halevy, and David Maier. From Databases to Dataspaces: A New Abstraction for Information Management. *SIGMOD Record*, 34, 2005.

[49] N. Fuhr and K. Großjohann. XIRQL: An XML Query Language Based on Information Retrieval Concepts. *ACM Transactions on Information Systems (TOIS)*, 2004.

[50] Google desktop. `http://desktop.google.com`.

[51] Gang Gou and Rada Chirkova. XML Query Processing: A Survey. Technical Report TR-2005-22, NC State University, 2005.

[52] Torsten Grust. Accelerating XPath Location Steps. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2002.

[53] Karl Anders Gyllstrom, Craig Soules, and Alistair Veitch. Confluence: Enhancing Contextual Desktop Search. In *Proceedings of the ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, 2007.

[54] Alon Halevy, Michael Franklin, and David Maier. Principles of Dataspace Systems. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems (PODS)*, 2006.

[55] Alan Halverson, Josef Burger, Leonidas Galanis, Ameet Kini, Rajasekar Krishnamurthy, Ajith Nagaraja Rao, Feng Tian, Stratis D. Viglas, Yuan Wang, Jeffrey F. Naughton, and David J. DeWitt. Mixed Mode XML Query Processing. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2003.

[56] INEX. `http://inex.is.informatik.uni-duisburg.de/`.

[57] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V. S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. TIMBER: A Native XML Database. *The VLDB Journal*, 11, 2002.

[58] William Jones, Ammy Jiranida Phuwanartnurak, Rajdeep Gill, and Harry Bruce. Don't Take My Folders Away!: Organizing Personal Information to Get Things Done. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, 2005.

[59] Yaron Kanza. Flexible Queries over Semistructured Data. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems (PODS)*, 2001.

[60] David R. Karger, Karun Bakshi, David Huynh, Dennis Quan, and Vineet Sinha. Haystack: A Customizable General-Purpose Information Management Tool for End Users of Semistructured Data. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, January 2005.

[61] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F Naughton, and Raghu Ramakrishnan. On the Integration of Structure Indexes and Inverted Lists. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[62] Pim Keizer. Indexing Methods for XML Documents, 2006.

[63] Christopher S.G. Khoo, Brendan Luyt, Caroline Ee, Jamila Osman, Hui-Hui Lim, and Sally Yong. How Users Organize Electronic Files on Their Workstations in the Office Environment: A Preliminary Study of Personal Information Organization Behaviour. *Information Research*, 2007.

[64] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2001.

[65] Yukun Li and Xiaofeng Meng. Research on Personal Dataspace Management. In *Proceedings of the SIGMOD PhD workshop on Innovative database research (IDAR)*, 2008.

[66] Catherine C. Marshall. How People Manage Information over a Lifetime.

[67] Yasuko Matsubara and Ichiro Kobayashi. Development of A Desktop Search System Using Correlation between User's Schedule and Data in a Computer. In *Proceedings of the International Conferences on Web Intelligence and Intelligent Agent Technology (WI-IATW)*, 2007.

[68] Shankar Pal, Istvan Cseri, Oliver Seeliger, Gideon Schaller, Leo Giakoumakis, and Vasili Zolotov. Indexing XML Data Stored in a Relational Database. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2004.

[69] Christopher Peery, Francisco Matias Cuenca-Acuna, Richard P. Martin, and Thu D. Nguyen. Wayfinder: Navigating and Sharing Information in a Decentralized World. In *Proceedings of Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, August 2004.

[70] Personal Information Management: `http://en.wikipedia.org/wiki/Personal_information_management`.

[71] Neoklis Polyzotis, Minos Garofalakis, and Yannis Ioannidis. Approximate XML Query Answers. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[72] Sujeet Pradhan. Towards a Novel Desktop Search Technique. In *Proceedings of the International Conference on Database and Expert Systems Applications*, 2007.

[73] Query expansion: `http://en.wikipedia.org/wiki/Query_expansion`.

[74] Torsten Schlieder. Schema-Driven Evaluation of Approximate Tree-Pattern Queries. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2002.

[75] Jayavel Shanmugasundaram, Rajasekar Krishnamurthy, Igor Tatarinov, Eugene Shekita, Efstratios Viglas, Jerry Kiernan, and Jeffrey Naughton. A General Technique for Querying XML Documents using a Relational Database System. *SIGMOD Record*, 30, 2001.

[76] Jayavel Shanmugasundaram, Kristin Tufte, Gang He, Chun Zhang, David DeWitt, and Jeffrey Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1999.

[77] Dongwook Shin, Hyuncheol Jang, and Honglan Jin. Bus: an effective indexing and retrieval scheme in structured documents. In *Proceedings of the Internaional Conference on Digital Libraries*, 1998.

[78] Craig A. N. Soules and Gregory R. Ganger. Connections: Using Context to Enhance File Search. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2005.

[79] Apple MAC OS X Spotlight. `http://www.apple.com/macosx/features/spotlight`.

[80] Jaime Teevan, Christine Alvarado, Mark Ackerman, and David Karger. The Perfect Search Engine is Not Enough: A Study of Orienteering Behavior in Directed Search. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, 2004.

[81] Anja Theobald and Gerhard Weikum. The Index-based XXL Search Engine for Querying XML Data with Relevance Ranking. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2002.

[82] Marcos Antonio Vaz Salles, Jens-Peter Dittrich, Shant Kirakos Karakashian, Olivier René Girard, and Lukas Blunschi. iTrails: Pay-as-you-go Information Integration in Dataspaces. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2007.

[83] Felix Weigel, Holger Meuss, Klaus U. Schulz, and Francois Bry. Content and Structure in Indexing and Ranking XML. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2004.

[84] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, Inc, 1999.

[85] Jens E. Wolff, Holger Flörke, and Armin B. Cremers. Searching and Browsing Collections of Structural Information. In *Proceedings of IEEE Advances in Digital Libraries*, 2000.

[86] WordNet. `http://www.cogsci.princeton.edu/~wn`.

[87] Xiaoying Wu, Stefanos Souldatos, Dimitri Theodoratos, Theodore Dalamagas, and Timos Sellis. Efficient Evaluation of Generalized Path Pattern Queries on XML Data. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, 2008.

[88] An XML Query Language. `http://www.w3.org/TR/xquery`.

[89] Z. Xu, M. Karlsson, C. Tang, and C. Karamanolis. Towards a Semantic-Aware File Store. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, May 2003.

[90] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel: A Path-Based Approach to Storage and Retrieval of XML, 2001.

[91] Yuqing Wu Yuwu and Yuqing Wu. Structural Join Order Selection for XML Query Optimization. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2003.

[92] Chun Zhang, Jeffrey Naughton, David Dewitt, and Qiong Luo. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2001.

Vita

Wei Wang

1994	**B.S.E. in Computer Science and Engineering, Zhejiang University, Hangzhou, China**
1997	**M.E. in Pattern Recognition and Artificial Intelligence, Institute of Automation, Chinese Academy of Sciences, Beijing, China**
2000	**M.S. in Computer Science, Rutgers University, New Brunswick, New Jersey, USA**
2010	**Ph.D. in Computer Science, Rutgers University, New Brunswick, New Jersey, USA**

Selected Publications

2008	"Multi-Dimensional Search for Personal Information Management Systems". C. Peery, W. Wang, A. Marian, and T. D. Nguyen. In Proceedings of the International Conference on Extending Database Technology (EDBT), March 2008.
2008	"Fuzzy Multi-Dimensional Search in the Wayfinder File System". C. Peery, W. Wang, A. Marian, and T. D. Nguyen. In Proceedings of the International Conference on Data Engineering (ICDE - Demo Track), April 2008.
2009	"Flexible Querying of Personal Information". A. Marian and W. Wang, IEEE Data Engineering Bulletin, 32(2): 20-27, 2009.