

# Multi-Dimensional Search for Personal Information Management Systems

Christopher Peery, Wei Wang, Amélie Marian, Thu D. Nguyen  
Department of Computer Science, Rutgers University  
110 Frelinghuysen Rd, Piscataway, NJ 08854, USA  
{peery, ww, amelie, tdnguyen}@cs.rutgers.edu

## ABSTRACT

With the explosion in the amount of semi-structured data users access and store in personal information management systems, there is a need for complex search tools to retrieve often very heterogeneous data in a simple and efficient way. Existing tools usually index text content, allowing for some IR-style 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. We propose a novel multi-dimensional approach to semi-structured data searches in personal information management systems by allowing users to provide fuzzy structure and metadata conditions in addition to keyword conditions. Our techniques provide a complex query interface that is more comprehensive than content-only searches as it considers three query dimensions (content, structure, metadata) in the search. We propose techniques to individually score each dimension, as well as a framework to integrate the three dimension scores into a meaningful unified score. Our work is integrated in Wayfinder, an existing fully-functioning file system. We perform a thorough experimental evaluation of our techniques to show the effect of approximating individual dimensions on the overall scores and ranks of files, as well as on query performance. Our experiments show that our scoring strategy adequately takes into account the approximation in each dimension to efficiently evaluate fuzzy multi-dimensional queries. In addition, fuzzy query conditions in non-content dimensions can significantly improve scoring (and thus ranking) accuracy.

## 1. INTRODUCTION

Dataspaces, large collections of heterogeneous data, and personal information management systems have recently received a lot of attention in the Database and Information Retrieval (IR) communities [8, 12, 15]. Unlike the very structured DBMSs, or the relatively unstructured document model from IR, dataspace allow for flexible management of data with varying degree of structure, and coming from possibly different sources. In addition, potentially useful metadata information can be stored alongside data. Allowing for powerful and flexible search in such information management systems is a critical issue; with the explosion in the amount

of data users access and store there is a need for complex search tools to access often very heterogeneous data in a simple and efficient way. Numerous third-party 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* [17] and *Spotlight* [23]. 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.

In this paper, we present scoring and searching techniques to access files in a personal information system scenario. While the research community has considered the problem of search in similar scenarios [15, 8, 12], as is the case with commercial tools, these work focus on IR-style keyword queries and use other system information, such as metadata, to guide the keyword-based search. Unfortunately, simple keyword-only searches are often insufficient, as illustrated by the following example:

*EXAMPLE 1. Consider a user saving personal information on a computing device. In addition, some of the data available on the device may come from external sources (such as other users in a network setting) and therefore may not be familiar to the user. Alongside the data, the system may store (a potentially large amount of) metadata information (e.g., access time, file type), as well as some navigational structure information (e.g., directory structure).*

*In such a scenario, it is possible to ask the query: Find a pdf file created on March 21, 2007 that contains the words “proposal draft.”*

*Keyword-based search tools would answer this query by returning all files of type \*.pdf created on 03/21/2007 (filtering conditions) that have content similar to “proposal draft” (ranking expression), ranked based on how close the content matches the text “proposal draft” using some underlying text scoring mechanism. These tools would therefore miss any relevant file that do not strictly adhere to the date and file type filtering conditions; for example, \*.tex documents created on 03/19/2007 would not be returned even if their content match the query keywords.*

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 Example 1 query, the user might not remember the exact creation date of the file of interest—or may not be the original creator of the file—but remembers that it was created *around* 03/21/2007. Similarly, the user might be primarily interested in files of type \*.pdf but might also want to consider relevant files of different but related types (e.g., \*.tex, or \*.txt). In this case, the date and file type

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EDBT’08, March 25–30, 2008, Nantes, France.

Copyright 2008 ACM 978-1-59593-926-5/08/0003 ...\$5.00.

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. 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.

In this paper, we propose a novel 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, 2, 6, 10] 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 our work could be extended to a variety of dataspace applications and queries, we focus on a file search scenario in this paper. 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 paper, we make the following contributions:

- We propose *IDF*-based scoring mechanisms for individual query dimensions (content, metadata, structure). Our scoring techniques take into account the specificity of each dimension, as well as the data distribution, to efficiently assign relevance scores to query answers.
- We propose a framework to combine individual dimension scores into a unified multi-dimensional score.
- We adapt existing top-*k* query processing algorithms to our scenario.
- We integrated our work into Wayfinder [20], an existing fully-functioning file system.
- We evaluated our scoring framework experimentally and show that our *IDF*-based scoring approach preserves the specificity of each dimension and that our unified scoring framework accurately captures multi-dimensional query flexibility. We report on the impact of multi-dimensional scoring on the query answers returned to the user and show that our techniques result in good overall query performance.

The rest of the paper is organized as follows: in Section 2, we present our multi-dimensional scoring framework. Section 3 describes the overall architecture of our system and the algorithms we use to aggregate scores and return the best answers to the queries. We present our experimental results in Section 4. Finally, we discuss related work in Section 5 and conclude in Section 6.

## 2. 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 [24]. The content *IDF* scoring strategy has been widely adopted in IR systems as it considers the data distribution 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 dimension 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 4.

We first give a brief overview of our query model (Section 2.1), we then present our *IDF*-based scoring strategies for each dimension: content (Section 2.2), metadata (Section 2.3), and structure (Section 2.4). Finally, we show how we aggregate scores across dimension in Section 2.5.

### 2.1 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 a simplified version of XQuery [25] as our query language. For instance, the query for Example 1 would be expressed as follows:

```
FOR $i IN /File[FileSysMetadata/FileDate
              = '03/21/07']
  FOR $j IN /File[ContentSummary/WordInfo/Term
                  = 'proposal'
                  AND ContentSummary/WordInfo/Term
                  = 'draft']
    FOR $m IN /File[FileSysMetadata/FileType
                    = 'pdf']
      WHERE $i/@fileID = $j/@fileID
      AND $i/@fileID = $m/@fileID
      RETURN $i/fileName
```

An answer to the query is a file that is relevant to one or more of the query parameters. 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 to each query parameter based on how close the match is to the condition (exact matches for filtering conditions have a score of 1).

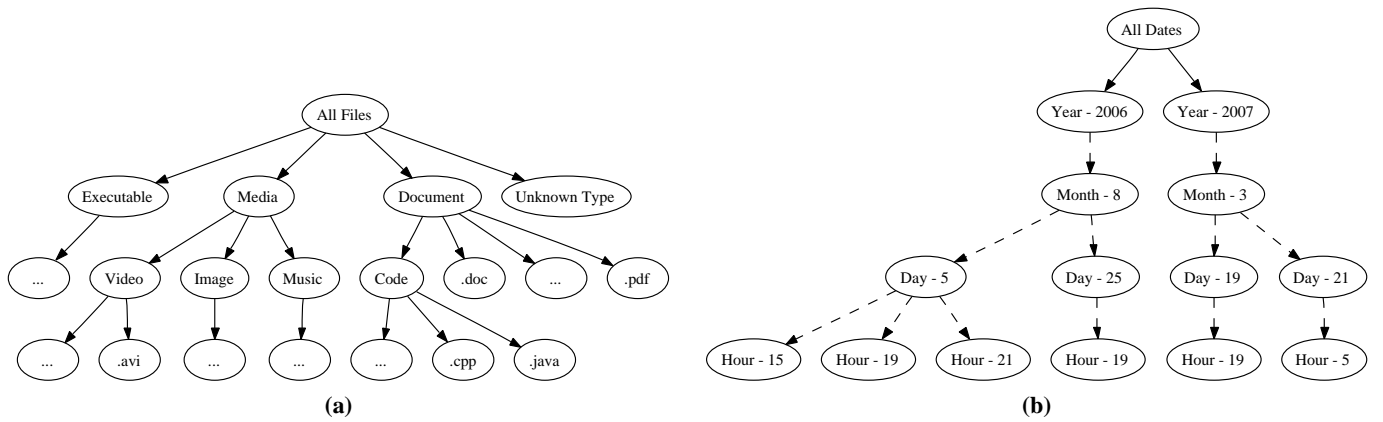
In the rest of this section, we discuss our scoring strategies for approximating query matches in each dimension.

### 2.2 Scoring Content

We use standard indexing structures and scoring mechanisms from the IR literature [24] for query conditions involving text. Specifically, we have implemented inverted indices for identifying files containing query terms and use a  $TF \cdot IDF$  scoring strategy for scoring text conditions, as described below.

**DEFINITION 1** (CONTENT  $TF \cdot IDF$  SCORE OF A FILE). *For a given keyword query,  $Q$ , consisting of the terms  $t_1, t_2, \dots, t_n$ , the content score of a file  $F$ , with respect to  $Q$  is computed as:*

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



**Figure 1: Fragments of the indexing DAGs for (a) file type (extension) metadata, and (b) file date metadata. Solid lines represent parent-child relationships. Dotted lines represent ancestor-descendant relationships, with intermediate nodes removed for simplicity of presentation.**

with

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

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.

## 2.3 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.

In this section we discuss our scoring strategies for metadata information. We develop the concept of *metadata relaxation* to score and retrieve approximate matches to metadata query conditions.

### 2.3.1 DAG Representation of Metadata Relaxations

We use a DAG indexing structure to support the scoring of relaxations on metadata conditions. In particular, we construct a DAG-based index for each type of searchable metadata to support both the retrieval and the scoring of results. In these indexes, possible metadata values that files can have are stored in the leaves of the DAG. Internal nodes of the DAG then represents progressive hierarchical generalizations of their children.

For example, Figure 1 represents (subgraphs of) the DAGs associated with file types (Figure 1(a)) and file dates (Figure 1(b)). For the file type DAG, each leaf represents a specific file type (e.g., `.doc` and `.pdf`) and contains a count as well as references to all files of that type. Each internal node represents a more general file type that is a union of the types of its children (e.g., `Media` is the union of `Video`, `Image`, and `Music` types) and thus a relaxation of its descendants. Correspondingly, each internal node contains the sum of the

file counts of its descendant leaves. Note that the count maintained at each internal node is thus guaranteed to be greater than or equal to the count at any of its children. Similarly, in the DAG for file dates, individual timestamps are represented at the leaf node level and internal nodes represent larger time periods spanning those represented by their descendants. A similar DAG is built to represent file sizes.

### 2.3.2 Scoring Metadata Relaxations

As mentioned, the metadata DAG indexes are used for scoring. Our  $IDF$ -based framework requires the score of a file to depend on how many files match a given relaxation of a query condition. Given a specific metadata condition, the path from the matching leaf to the root of the DAG index for a metadata type represents all of the approximations that we can score for that condition. For instance, the query condition `FileType='pdf'` in Example 1 would exactly match the leaf `.pdf` in the DAG example of Figure 1(a). The leaf's ancestor nodes, `Documents` and `All Files`, represent approximate matches.

Continuing the example, our  $IDF$ -based scoring approach then scores matching files as follows. Files of type `.pdf` would have the highest score as they are exact matches to the query condition. Files of type `Document`, other than type `.pdf`, e.g., `.doc` and `Code`, would be assigned a lower score. Finally, files of type `All Files`, other than type `Document`, which would consist of all remaining files in the system, would be assigned yet a lower score. The latter two assigned scores are for the two approximations of the query condition as they are assigned to files that do not match the query condition exactly.

**DEFINITION 2 (METADATA CONDITION SCORE OF A FILE).** For a given metadata query,  $Q$ , consisting of a target value  $v_Q$  for a metadata condition  $C$ , the metadata score of a file  $F$ , with corresponding metadata value  $v_F$ , with respect to  $Q$  is computed as:

$$score_{MetaData}(Q, F, C) = \frac{\log\left(\frac{N}{|fileDesc(commonAnc(v_Q, v_F))|}\right)}{\log(N)}$$

where  $N$  is the total number of files,  $commonAnc(x, y)$  returns the closest common ancestor of nodes  $x$  and  $y$ , and  $fileDesc(x)$  returns the files that can be reached through the descendants of node  $x$  in the metadata DAG. The score is normalized by  $\log(N)$  so that a single perfect match would have the highest possible score

of 1. The most relaxed matches to the condition  $C$  will have a score of 0 as their closest common ancestor with  $v_Q$  is the root node of the DAG which contains all  $N$  files as its descendants.

Intuitively, we find the closest common ancestor of  $v_Q$  and  $v_F$  in the metadata DAG, and count the number of files that can be reached through descendants of this common ancestor. The higher this number is, the lower the score of  $F$  for  $Q$  will be as many other files share the same level of approximation with  $Q$  as  $F$ .

### 2.3.3 Aggregating Metadata Scores

For queries involving multiple metadata conditions (e.g., our example query, with a condition on date and a condition on filetype) the individual condition scores have to be aggregated to produce a unified metadata score.

We aggregate individual metadata scores by considering both the query and the document as vectors of dimension  $n$ , where  $n$  is the number of individual metadata conditions  $C_1, \dots, C_n$ . The document vector  $\vec{V}_F$  consists of the individual  $score_{MetaData}(Q, F, C_i)$  ( $1 \leq i \leq n$ ). The query vector  $\vec{V}_Q$  has value 1 (exact match) for each dimension. The unified metadata score is then the normalized length of the projection of the document vector on the query vector.

**DEFINITION 3 (METADATA SCORE OF A FILE).** For a given metadata query  $Q$  with corresponding query vector  $\vec{V}_Q$ , consisting of several metadata value conditions  $C_1, \dots, C_n$ , the metadata score of a file  $F$  with corresponding document vector  $\vec{V}_F$ , with respect to  $Q$  is computed as:

$$score_{MetaData}(Q, F) = \frac{\vec{V}_F \cdot \vec{V}_Q}{|\vec{V}_Q|}$$

Note that if only one metadata condition  $C$  is present in  $Q$ , then  $score_{MetaData}(Q, F) = score_{MetaData}(Q, F, C)$ .

## 2.4 Scoring Structure

Most users typically 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 [12]. However, users are notoriously bad at remembering where they stored a particular file or how the files are structured [11]. 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 is on directory path structure, our structure scoring techniques could easily be adapted for structural information within a document. Our structure scoring strategy is based on work on XML structural query relaxations [3, 5, 4]; as described below, we introduce new types of structural relaxations to handle the specific needs of user searches in a personal information management system.

### 2.4.1 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.

### 2.4.2 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 [3, 4, 5], 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 [3]. 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 [4, 5].

As in [4, 5], 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 exact 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*( $n$ ) 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  $*$  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 [4, 5]. In particular, node inversions significantly complicate and increase the number of possible query relaxations and require the introduction of the node group notation (Section 2.4.1) to represent path permutations.

### 2.4.3 DAG Representation of Structure Relaxations

As proposed in [4], 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 (Section 3). Figure 2 shows an example relaxation DAG, along with example *IDF* scores (see Section 2.4.4 for details), for the structure query condition *Personal/Ebooks/JackLondon*. 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 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 1. Matches for the exact query  $P/E/J$  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  $P/E/J$ :  $//*$  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 Function

---

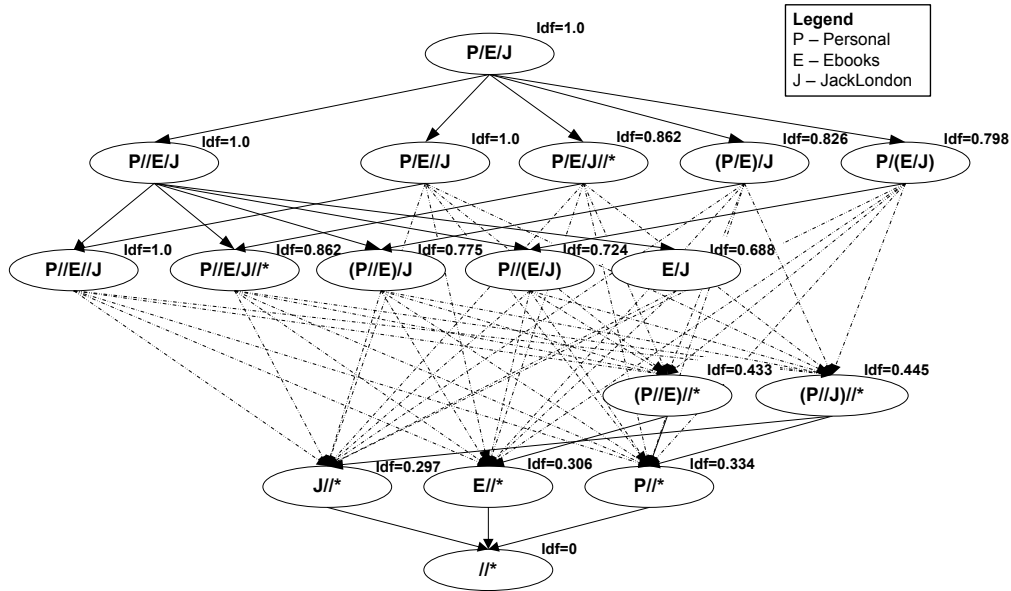
```

function buildPQDAG(currentDAGNode)
begin
  P = getQuery(currentDAGNode)
  for each edge e in P do
    if isParentChildEdge(e) then
      newDAGNode = getDAGNode(edgeGeneralize(e, P))
      {getDAGNode checks if a DAG node containing P with the edge
      generalization exists, and creates such a DAG node if it does not.}
    end if
  end for
  for each node or node group n in P do
    if not isWildcardNode(n) and parent(n) exists and
    not isRootNode(parent(n)) then
      newDAGNode =
        getDAGNode(invertNode(n, parent(n), P))
    end if
  end for
  for each node or node group n in P do
    if requireNodeDeletion(n) then
      if isNode(n) then
        newDAGNode = getDAGNode(nodeDeletion(n, P))
      else
        {n is a node group.}
        for each node m in n do
          newDAGNode =
            getDAGNode(nodeDeletion(m, P))
        end for
      end if
    end if
  end for
end

```

---

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



**Figure 2: The structure DAG for the structural query condition *Personal/Ebooks/JackLondon*. Solid lines represent parent-child relationships. Dotted lines represent ancestor-descendant relationships, with intermediate nodes removed for simplicity of presentation.**

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.

#### 2.4.4 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 [4].

**DEFINITION 4 (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:

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

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

Figure 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 2 path queries  $P/E/J$ ,  $P//E/J$ , and  $P//E//J$  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 4). Because answers to a path query  $P$  are contained in the set of answers to any relaxation of  $P$ , this implies that  $P/E/J$ ,  $P//E/J$ , and  $P//E//J$  have the same set of answers, i.e., these

relaxations do not result in more approximate matches. This observation is useful for our query processing algorithm (Section 3.2), as we can use this information to “skip” some DAG nodes while traversing the DAG.

**DEFINITION 5 (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:

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

where  $P'$  is any path query that is a relaxation of  $Q$  and  $F(D, P')$  is the set of all files that can be reached through  $P'$  and its extensions in  $D$ .

## 2.5 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.

The individual dimension scores are aggregated to produce the final score of a file for a query. We use a vector projection for the aggregation of multi-dimension scores, similar to the one we used for aggregating individual metadata condition scores (Section 2.3.3). 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:

$$\vec{V}_F = (score_{Content}(Q, F), score_{MetaData}(Q, F), score_{Structure}(Q, F))$$

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 6 (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:

$$\text{score}(Q, F) = \frac{\vec{V}_F \cdot \vec{V}_Q}{|\vec{V}_Q|}$$

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. IMPLEMENTATION

To experimentally evaluate our approach, we have implemented the above multi-dimensional scoring framework in the Wayfinder file system [20]. We have also implemented a top- $k$  query processing algorithm to efficiently find the top  $k$  relevant results using our scoring framework. This section briefly describes our implementation.

#### 3.1 Indexing Structures

We choose Wayfinder as the hosting platform for an implementation of our scoring framework mostly based on convenience. Wayfinder is a user-level file system, which makes it significantly easier to modify. In addition, Wayfinder already implements a  $TF \cdot IDF$ -based content search engine (along with the necessary indexes).

In this work, we have extended Wayfinder to include the following set of data tables and inverted indexes to support the processing of query conditions in the metadata and structural dimensions.

Metadata:

- A metadata table, where each row contains the typical meta data, e.g., owner, size, date of creation, etc., for each file in the system. (In current file systems, this information is typically dispersed throughout the file system, making it difficult to find files that match specific metadata query conditions.)
- Five indexes of the metadata table to support efficient searches on the size, type, date-of-creation, last-modified-time, and last-accessed time attributes. Each of these index corresponds to a DAG indexing structure similar to the ones presented for file type and file date in Section 2.3.

Structural:

- Two name-binding tables to hold “parent directory  $\rightarrow$  children” and “child  $\rightarrow$  parent directory” bindings.
- An inverted index that maps terms to full directory pathnames that contain these terms for quickly finding directories that match a structural query conditions and its relaxed forms.

Wayfinder is written in Java. Tables and indexes are persistently stored using the Berkeley DB [21]. Search queries are specified using a small subset of the XQuery language (Section 2.1).

#### 3.2 Query Processing

Our query model (Section 2.1) supports ranked retrieval of answers based on how closely they match a query. Numerous work on top- $k$  query processing has shown that evaluating all possible matches to return the  $k$  best answers is prohibitively expensive. In

our scenario, this would lead to scoring and ranking every single file in the system for each query. Several top- $k$  query processing techniques have been proposed in the Database community in recent years. We decided to use an existing and popular algorithm for evaluating top- $k$  answer: the Threshold Algorithm (TA) [14].

The focus of the current paper is on scoring strategies. Due to lack of space, we will not detail our implementation of query optimization strategies (although we report on experimental performance results in Section 4.6). However, we have implemented specific optimizations to the TA algorithm for our scenario, as well as efficient index traversal techniques to help speed up query execution. In particular, as observed in [4] for the XML scenario, the size of structure relaxation DAGs is exponential to the size of the structure query. To avoid materializing large DAGs, we have implemented a lazy traversal of the DAG which only materializes those nodes that are needed during query processing. We plan to report on these optimizations and their impact on query performance in future work.

### 4. EXPERIMENTAL EVALUATION

We now 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.

#### 4.1 Experimental Setting

**Platform.** All experiments were performed using the Wayfinder file system. 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 1.4.2 JVM.

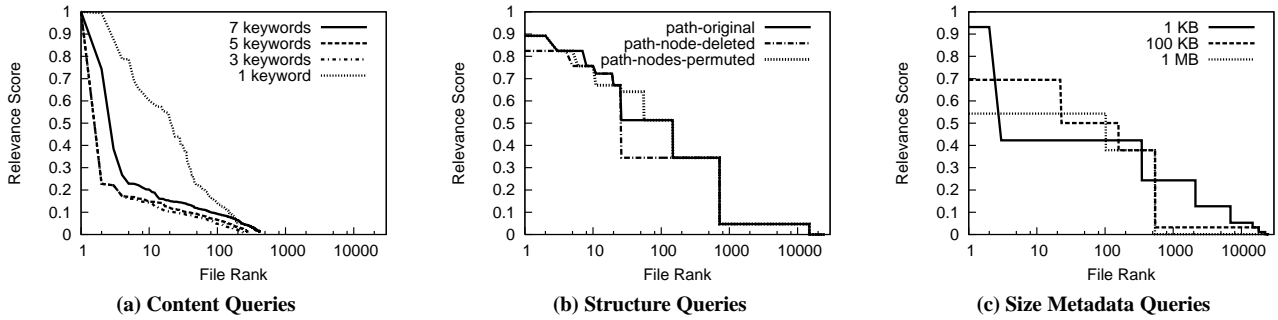
**Data Set.** As noted in [12], there is a lack of synthetic data sets and benchmarks to evaluate search over personal information management systems; therefore we used a real user data set comprised of (a representative subset of) files and directories from the working environment of one of the authors. This data set contained 24,926 files in 2,338 directories. 24% of this data set were multimedia files (e.g., music and pictures), 17% document files (e.g., pdf, text, and MS Office), 14% email messages,<sup>1</sup> and 12% source code files. The average directory depth was 3.4 with the longest being 9. On average, directories contained 11.6 sub-directories and files, with the largest—a folder containing emails—containing 1013. Wayfinder extracted 347,448 unique stemmed content terms.<sup>2</sup> File modification dates spanned 10 years. 75% of the files were smaller than 177 Kbytes, and 95% of the files were smaller than 4.1 Mbytes.

#### 4.2 Behaviors of Scoring Functions

We start by studying the behaviors of our scoring functions. Figure 3 plots the scores of all relevant files as a function of their ranking for three 1-dimensional queries targeting three different dimensions as follows. Figure 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 is based on  $TF \cdot IDF$  as opposed to just  $IDF$ . Figure 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

<sup>1</sup>Email messages are stored in the Maildir format in which each email is stored in a separate file.

<sup>2</sup>Content was extracted from MP3 music files using their ID3 tags.



**Figure 3: Relevance score plotted as a function of ranking for (a) four content queries (b) three structure queries, and (c) three size metadata queries**

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(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(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(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 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 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. We leave such exploration as future work.

### 4.3 Scores and Rankings for Approximate Answers

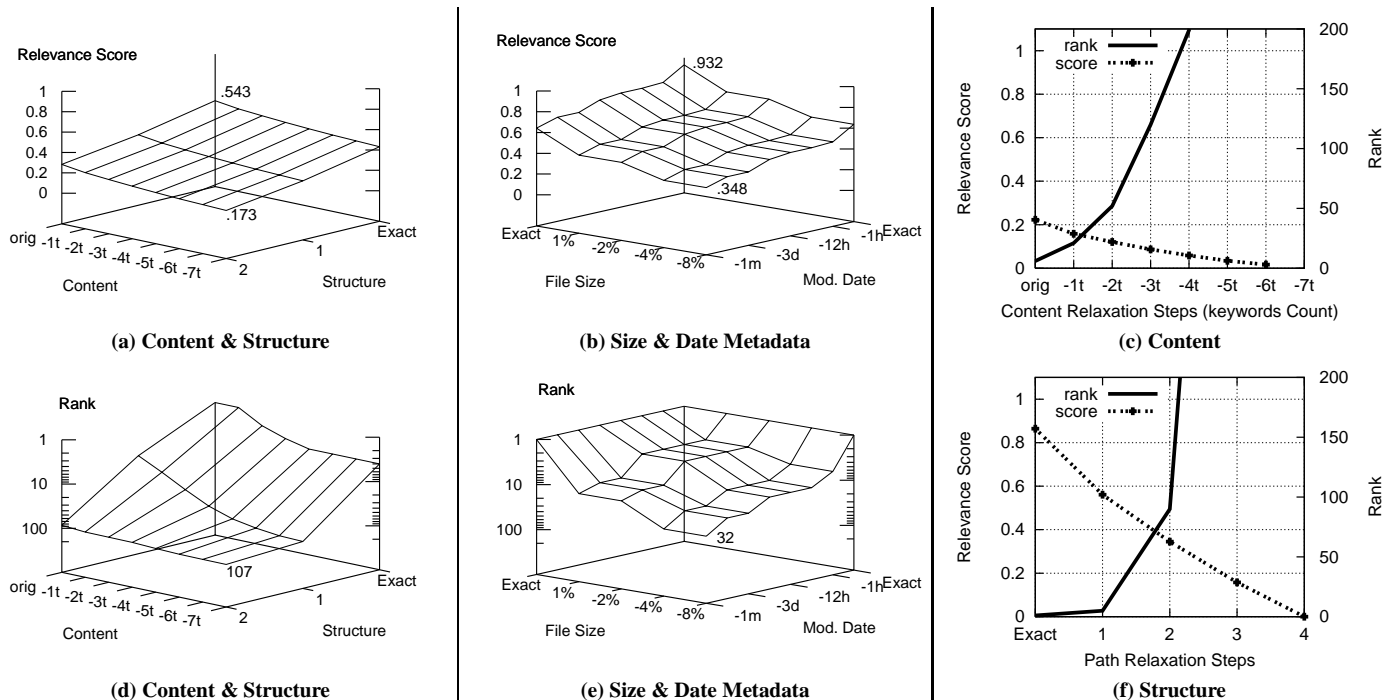
We now show how scoring (and the corresponding ranking) is affected by inaccurate query conditions. For this purpose, we choose a target file that is an exact match for the query conditions of a particular query, resulting in a high score and rank. We then modify the target file so that it is progressively 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 4(a-b) and (d-e) plot the score and rank, respectively, of the target file for two representative 2-dimensional queries covering the four dimensions of content, structure, size metadata, and (last modified) date metadata. 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. In the size metadata dimension, we relax the file by progressively decreasing its size (we also relax in the opposite direction but do not plot the results because they are not significantly different). In the date metadata dimension, we relax the file by progressively moving its date backward from the query condition.

It should be noted that in Figure 4, the target file is not the only exact match to the query condition in the structure, size, and date dimensions, leading to a relevance score of less than 1 in each of these dimensions 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 4(a-b) 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 4(c) and (f) for comparison purposes. Although the results are not shown here, scoring for 3-dimensional queries also display similar behaviors.

More interestingly, we observe that providing query conditions



**Figure 4:** Score (a-b) and rank (d-e) 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 target file plotted for (c) a content-only query and (f) a structure-only query.

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 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 4(d)). Similarly, if we provide information on the file’s size and date, even when the size is 8% off and the date is incorrect by 1 month, the target file is ranked 32nd (Figure 4(e)). Interestingly, sometimes inaccurate query conditions on one dimension do not affect ranking, as is shown in (Figure 4(e)) where a slight approximation in the query condition of one dimension, provided the other dimension is exact, still results in a rank of 1 since few exact matches to each individual dimension exist in the data set.

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.

#### 4.4 Impact of Flexible Multi-Dimensional Search

In the last section, we have 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 con-

ditions 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 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.<sup>3</sup>

<sup>3</sup>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.

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

**Table 1: The rank of a target file—the novel *Sea Wolf* by Jack London—returned by a set of related queries. 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/*. The column Ranked by Filtering indicates whether the target file would be considered as a relevant answer given today’s typical filtering approach.**

The results are quite promising. For example, in query Q15, just getting 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 an N for the **Ranked with Filtering** column. 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. We intend to validate this potential in more extensive user studies in future work.

## 4.5 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 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 [13]. The standard Spearman’s rho ( $\rho$ ) measures the distance between  $l_1$  and  $l_2$ , two permutations of the same list. The minimized Spearman’s rho ( $\rho_{min}$ ) 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 5 shows the  $\rho_{min}$  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 condi-

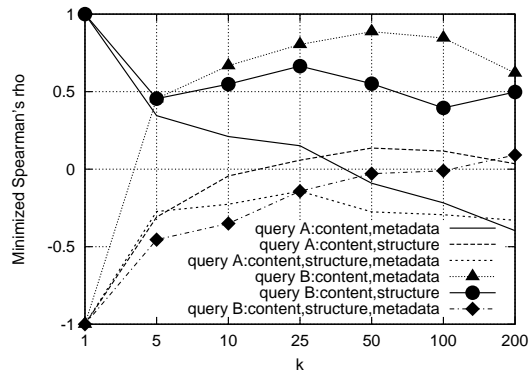


Figure 5:  $\rho_{min}$  value for various multi-dimensional queries as a function of  $k$ .

Query Dimensions	Total Query Time (ms)	
	In-Memory	Persistent
Content	68.29	224.05
Structure	75.85	466.11
Size	17.43	70.33
Date	18.31	80.53
Content and Size	261.81	1329.66
Date and Size	206.17	2194.17
Structure and Size	689.15	1875.08
Content and Structure	383.93	1115.25
Content, Structure and Size	1099.44	2887.83
Content, Structure, Size and Date	2078.78	6221.49

Table 2: Query performance for various single- and multi-dimensional queries for both in-memory indexes and persistent indexes.

tion 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.

## 4.6 Query Performance

While the focus of this paper is on a unified scoring framework and its impact on query results, query performance is an important aspect. We have implemented several top- $k$  query optimization techniques to speed up query evaluation (Section 3.2). Our techniques ensure that the correct top  $k$  answers for a query, according to our unified scoring framework, are returned to the user.

Table 2 shows the query performance of several single- and multi-dimensional top-200 queries using both completely in-memory indexes and persistent indexes. Recall that our persistent indexes were implemented using the Berkeley DB [21] via its Java API. Each number reported is an average of 50 query evaluations.

Immediately noticeable are the larger times for both content and structure when compared to either size and date. The large content times result from our current unoptimized index design. Processing a query for each term currently requires the retrieval of the entire list of all files that contain that term. For structure, the larger times result from constructing and evaluating the structural DAG at run time.

The increases in time between the in-memory and persistent index stems from the need to read data from the Berkeley database. We have implemented several simple caching mechanisms to minimize these accesses. We believe, however, significant opportunities for optimization remain.

The difference in times between one dimension and the multi-dimensional searches is largely due to the overhead of top- $k$  processing. While it may be cheap to access the top results for a single dimension using sorted indexes, a multi-dimensional search may require to access (via more expensive random accesses) files that have low scores in one or more dimensions.

We are currently investigating various methods to further improve performance as future work. Among these are more aggressive caching techniques to further minimize access to disk, adjustments to our top- $k$  algorithm that will reduce its computational cost, and probabilistic evaluation of the structural DAG. Our results show reasonable query response time. Given that these measurements were taken in an early, mostly unoptimized prototype, we believe our fuzzy multi-dimensional scoring approach is practical for implementation in real systems.

## 5. RELATED WORK

Several works have focused on the user perspective of personal information management [8, 19]. These works allow users to organize personal data semantically by creating associations between files or data entities and then leveraging these associations to enhance search.

Other works [12, 26] address information management by proposing generic data models for heterogeneous and evolving information. The works are aimed at providing users with generic and flexible data models to accessing and storing information beyond what is supported in traditional file system. Instead, we focus on querying information that is already present in the file system. An interesting future direction would be to include entity associations in our search and scoring framework.

Other file system related projects have tried to enhance the quality of search within file system by leverage the context in which information is accessed to find related information [22] or by altering the model of the file system to a more object-orientated database system [7]. These differ from ours in that they 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.

Recently there has been a surge in projects attempting to improve Desktop search [23, 17]. These projects provide search capabilities over content and then employ other pieces of information such as size, date, or types as filtering conditions.

The INitiative for the Evaluation of XML retrieval (INEX) [18] 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. As part of the INEX effort, XIRQL [16] 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 [9] 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,

XML structural query relaxations have been discussed in [3, 5, 4]. These works focus on the XML documents. In contrast, we are looking at personal information file system which have more variations in the types 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 [4], or the relaxed query containment condition [5, 4]. Our techniques differ from these work as we consider relaxations specific to path queries and ignore twig queries. In particular, we introduce specific relaxations for path queries in a directory-based file system.

## 6. CONCLUSION AND FUTURE WORK

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 and on query performance. We are currently working on query processing optimization techniques to improve the performance of our system.

In the future, we plan to extend our scoring dimensions to include file context information, in the spirit of [22]. We also plan to qualitatively evaluate our scoring methodology through user evaluations.

## 7. REFERENCES

- [1] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated Ranking of Database Query Results. In *Proc. of the Conference on Innovative Data Systems Research (CIDR)*, 2003.
- [2] S. Amer-Yahia, P. Case, T. Rölleke, J. Shanmugasundaram, and G. Weikum. Report on the DB/IR panel at SIGMOD 2005. *SIGMOD Record*, 2005.
- [3] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree Pattern Relaxation. In *Proc. of the International Conference on Extending Database Technology (EDBT)*, 2002.
- [4] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman. Structure and Content Scoring for XML. In *Proc. of the International Conference on Very Large Databases (VLDB)*, 2005.
- [5] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. FlexPath: Flexible Structure and Full-Text Querying for XML. In *Proc. of the ACM International Conference on Management of Data (SIGMOD)*, 2004.
- [6] R. A. Baeza-Yates and M. P. Consens. The Continued Saga of DB-IR Integration. In *Proc. of the International Conference on Very Large Databases (VLDB)*, 2004.
- [7] C. M. Bowman, C. Dharap, M. Baruah, B. Camargo, and S. Potti. A File System for Information Management. In *Proc. of the International Conference on Intelligent Information Management Systems (ISMM)*, 1994.
- [8] Y. Cai, X. L. Dong, A. Halevy, J. M. Liu, and J. Madhavan. Personal Information Management with SEMEX. In *Proc. of the ACM International Conference on Management of Data (SIGMOD)*, 2005.
- [9] D. Carmel, Y. S. Maarek, M. Mandelbrod, Y. Mass, and A. Soffer. Searching XML Documents via XML Fragments. In *Proc. of the ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, 2003.
- [10] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR Technologies: What is the Sound of One Hand Clapping? In *Proc. of the Conference on Innovative Data Systems Research (CIDR)*, 2005.
- [11] W. B. Croft, P. Krovetz, and H. Turtle. Interactive Retrieval of Complex Documents. *Information Processing and Management*, 1990.
- [12] J.-P. Dittrich and M. A. V. Salles. iDM: A Unified and Versatile Data Model for Personal Dataspace Management. In *Proc. of the International Conference on Very Large Databases (VLDB)*, 2006.
- [13] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM Journal on Discrete Mathematics*, 2003.
- [14] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. *Journal of Computer and System Sciences*, 2003.
- [15] M. Franklin, A. Halevy, and D. Maier. From Databases to Dataspace: A New Abstraction for Information Management. *SIGMOD Record*, 2005.
- [16] N. Fuhr and K. Großjohann. XIRQL: An XML Query Language Based on Information Retrieval Concepts. *ACM Transactions on Information Systems (TOIS)*, 2004.
- [17] Google desktop. <http://desktop.google.com>.
- [18] INEX. <http://inex.is.informatik.uni-duisburg.de/>.
- [19] D. R. Karger, K. Bakshi, D. Huynh, D. Quan, and V. Sinha. Haystack: A Customizable General-Purpose Information Management Tool for End Users of Semistructured Data. In *Proc. of the Conference on Innovative Data Systems Research (CIDR)*, 2005.
- [20] C. Peery, F. M. Cuenca-Acuna, R. P. Martin, and T. D. Nguyen. Wayfinder: Navigating and Sharing Information in a Decentralized World. In *Proc. of Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, 2004.
- [21] Sleepycat Software. Berkeley DB. <http://www.sleepycat.com/>.
- [22] C. A. N. Soules and G. R. Ganger. Connections: Using Context to Enhance File Search. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*, 2005.
- [23] Apple MAC OS X spotlight. <http://www.apple.com/macosx/features/spotlight>.
- [24] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Inc, 1999.
- [25] An XML Query Language. <http://www.w3.org/TR/xquery/>.
- [26] Z. Xu, M. Karlsson, C. Tang, and C. Karamanolis. Towards a Semantic-Aware File Store. In *Proc. of the Workshop on Hot Topics in Operating Systems (HotOS)*, May 2003.