Efficient Multi-Dimensional Query Processing in Personal Information Management Systems

Wei Wang, Christopher Peery, Amélie Marian, Thu D. Nguyen
{*ww, peery, amelie, tdnguyen*}*@cs.rutgers.edu*

Technical Report DCS-TR-627
Department of Computer Science, Rutgers University
110 Frelinghuysen Rd, Piscataway, NJ 08854

April 3, 2008

Abstract

The relentless growth in capacity and dropping price of storage are driving an explosion in the amount of information users are collecting and storing in personal information management systems. This explosion of information has led to a critical need for complex search tools to access often very heterogeneous data in a simple and efficient manner. Such tools should provide both high-quality flexible scoring mechanisms and efficient query processing capabilities. In this paper, we focus on indexes and algorithms to efficiently identify the most relevant files that match multi-dimensional queries comprised of relaxed content, metadata, and structure conditions. We also adapted existing top-k query strategies to our specific scenario. Our work is integrated in Wayfinder, an existing fully functioning file system. We perform a thorough experimental evaluation of our file search techniques and show that our query processing strategies exhibit good behavior across all dimensions, resulting in good overall query performance and good scalability.

1 Introduction

The amount of data stored in personal information management systems is rapidly increasing, following the relentless growth in capacity and the dropping price per byte of storage in personal computing systems. 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. Such tools should 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 file systems, such as the commercial tools Google Desktop [14] 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 the Information Retrieval (IR) community—but only consider structure (e.g., file directory) and metadata (e.g., date, file type) as *filtering* conditions. Recently, the research community has turned its focus on search over to Personal Information and Dataspaces [8, 10, 12], which consist of heterogeneous data collections. However, as is the case with commercial file system search tools, these works focus on IR-style keyword queries and use other system information only to guide the keyword-based search.

Keyword-only searches are often insufficient, as illustrated by the following example:

Example 1 *Consider a user saving personal information in the file system of a personal computing device. In addition to the actual file content, structural location information (e.g., directory structure) and a potentially large amount of metadata information (e.g., access time, file type) are also stored by the file system.*

In such a scenario, the user might want to ask the query:

```
[filetype=*.doc AND
```

```
createdDate=03/21/2007 AND
content="proposal draft" AND
structure=/docs/Wayfinder/proposals]
```

Current tools would answer this query by returning all files of type *.doc *created on 03/21/2007 under the directory* /docs/Wayfinder/proposals *(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. 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,* *.tex *documents created on 03/19/2007, or relevant files in the directory* /archive/proposals/Wayfinder *would not be returned.*

In previous work [19, 20], we presented a scoring framework that considers relaxed query conditions on several query dimensions. In these previous 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. For instance, for the Example 1 query, the user might not remember the exact creation date of the file of interest but remembers that it was created *around* 03/21/2007. Similarly, the user might be primarily interested in files of type **.doc* but might also want to consider relevant files of different but related types (e.g. **.tex* or **.txt*). Finally, the user might misremember the directory path under which the file was stored. In this case, by using the date, size, and structure conditions not as filtering conditions but as part of the ranking conditions of the query, our scoring techniques ensure that the best answers are returned as part of the search result. Our approach (summarized in Section 2) 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 [19] and [20], 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, we propose efficient indexes to identify and score query answers. The structure dimension, which exhibits complex relaxations such as permutations, is particularly challenging because: (1) 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; and (2) the index structures and algorithms should scale to handle a possibly large number of approximate matches.

Thus, in this paper, 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.

We make the following contributions:

- We propose efficient index structures and index construction algorithms for computing scores in the metadata and structure dimensions (Section 3.2 and 3.3).
- We present optimizations to improve access to the structure dimension index. 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 (Section 4.2 and 4.3).
- We perform a thorough experimental evaluation of our file search techniques and show that our query processing strategies result in good overall query performance and good scalability. (We implemented all proposed indexing structures and algorithms in a fully functioning file system called Wayfinder [18].) We empirically demonstrate the effect of our optimizations on query processing time and show that they drastically improve query efficiency (Section 5).

The rest of the paper is organized as follows. In Section 2, we give a high level overview of our query model and summarize our previous work on the unified scoring framework. In Section 3, we discuss our choice of top-k query processing algorithm and present our novel static indexing structures and score evaluation techniques. Section 4 describes the main contributions of the paper:

dynamic indexing structures and index construction algorithms for structural relaxations. In Section 5, we present our experimental results. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Query Model

Our system currently supports relaxed query conditions on three dimensions: *content* for conditions on the text content of files, *metadata* for conditions on the system information about files, and *structure* for conditions on the directory paths to access files. We internally represent files and their associated metadata and structure information as XML documents. Our relaxed queries can then be viewed as simple XQuery [27] expressions.

Any file that is relevant to one or more of the query conditions is then a potential answer for the query; it gets assigned a score for each dimension based on how close it matches the corresponding query condition. Scores across multiple dimensions are unified into a single overall score for ranking of answers.

Our scoring strategy was presented in [20]. Our approach is based on an *IDF*-based interpretation of scores, as described below. For each query condition, we score files based on the *least relaxed* form of the query that each file matches. Scoring along all dimensions is uniformly *IDF*-based which permits us to meaningfully aggregate multiple single-dimensional scores into a unified multi-dimensional score.

2.1 Content

We use standard IR relaxation and scoring techniques for content query conditions [26]. Specifically, we adopt the *TF·IDF* scoring formulas from Lucene [4], a state-of-the-art keyword search tool. These formulas are as follows:

$$score_c(Q, f) = \sum_{t \in Q} \frac{score_{c,tf}(t, f) \times score_{c,idf}(t)}{NormLength(f)} \quad (1)$$

$$score_{c,tf}(t, f) = \sqrt{\textit{No. times t occurs in f}} \quad (2)$$

$$score_{c,idf}(t) = 1 + \log(\frac{N}{1 + N_t}) \quad (3)$$

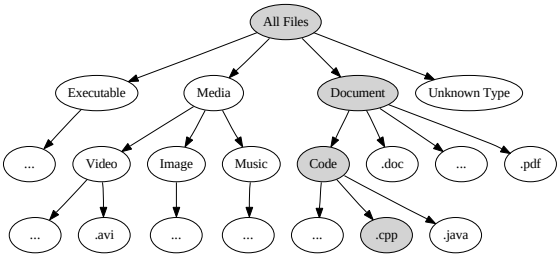


Figure 1: Fragment of the relaxation DAG for file type (extension) metadata.

where Q is the content query condition, f is the file being scored, N is the total number of files, N_t is the number of files containing the term t, and $NormLength(f)$ is a normalizing factor that is a function of f's length.[1] Note that relaxation is an integral part of the above formulas since they score all files that contain a subset of the terms in the query condition.

2.2 Metadata

We introduce a hierarchical relaxation approach for each type of searchable metadata to support scoring. For example, Figure 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 4 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

[1] See http://lucene.apache.org/java/2_4_0/scoring.html.

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:

$$score_{MetaData}(Q, f) = \frac{log(\frac{N}{nFiles(commonAnc(n_M, n_f))})}{log(N)} \quad (4)$$

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.

2.3 Structure

Our structure scoring strategy extends prior work on XML structural query relaxations [2, 3]. In particular, we use several types of structural relaxations, some of which were not considered in [2, 3], to handle the specific needs of user searches in a file system. Assuming that structure query conditions are given as pathnames, these relaxations are:

- **Edge Generalization** is used to relax a parent-child relationship to an ancestor-descendant relationship. For example, applying edge generalization to $/a/b$ would result in $/a//b$.
- **Path Extension** is used to extend a path P such that all files within the directory subtree rooted at P can be considered as answers. For example, applying path extension to $/a/b$ would result in $/a/b//*$.
- **Node Deletion** is used to drop a node from a path. For example, applying node deletion on b from $/a/b/c$ would result in $/a//c$.
- **Node Inversion** is used to permute nodes within a path. For example, applying node inversion on b and c from $/a/b/c$ would result in $/a/(b/c)$, allowing for both the original query condition as well as $/a/c/b$. We call the (b/c) part of the relaxed condition $/a/(b/c)$ a node group.

Note that the above relaxations can be applied to the original query condition as well as relaxed versions of the original condition.

We then say that a file *matches* a (possibly relaxed) query condition if all structural relationships between the condition's components are preserved in the file's parent directory. For example, the file $/a/b/c/f$ matches the condition $/a//c$ because the parent-child relationship between / and a and the ancestor-descendant relationship between a and c are preserved in the directory $/a/b/c$.

Finally, given a directory structure D and structure query Q, the structure score of a file f with respect to Q is computed as:

$$score_{Structure}(Q, f) = \max_{P \in R(Q)} \{score_{idf}(P) | f \in F(D, P)\} \quad (5)$$

$$score_{idf}(P) = \frac{log(\frac{N}{N_P})}{log(N)}, \quad N_P = |F(D, P)| \quad (6)$$

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 P's extension in D, and N is the total number of files in D. Note that the set of all possible relaxations of a query is query-dependent and so has to be generated at query time; the generation of these relaxations and efficient scoring of matching files is a major topic of Section 4.

2.4 Score Aggregation

We aggregate the above single-dimensional scores into a unified multi-dimensional score to provide a unified ranking of files relevant to a multi-dimensional query. To do this, we construct a query vector, $\vec{V_Q}$, having a value of 1 (exact match) for each dimension and a file vector, $\vec{V_F}$, consisting of the single-dimensional scores of file F with

respect to query Q. (Scores for the content dimension is normalized against the highest score for that query condition to get values in the range $[0, 1]$.) We then compute the projection of $\vec{V_F}$ onto $\vec{V_Q}$ and the length of the resulting vector is used as the aggregated score of file F. In its current form, this is simply a linear combination of the component scores with equal weighting. The vector projection method, however, provides a framework for future exploration of more complex aggregations.

3 Efficient Top-K Query Processing

While [19] and [20] focus on the flexible multi-dimensional scoring model and its impact on search result quality, the focus of the current paper 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) [11] 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.

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 3.1), metadata (Section 3.2), and structure (Section 3.3). In addition, we present access methods to support both sorted and random access on the content (Section 3.1) and metadata (Section 3.2) dimensions. We will 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.

3.1 Evaluating Content Scores

As mentioned in Section 2.1, we use existing *TF · 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.

3.2 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 2.2). 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 2.2). This is implemented as a simple DAG traversal algorithm. Random accesses for queries with multiple metadata conditions require the traversal of

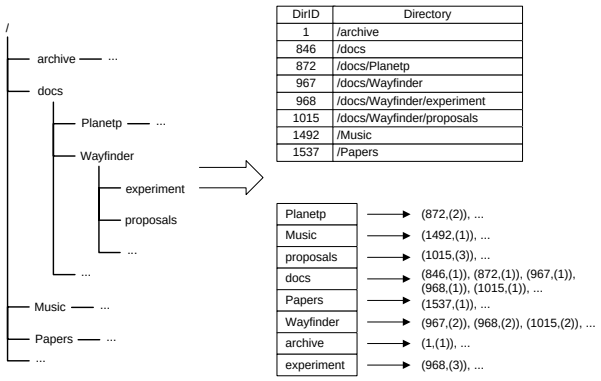


Figure 2: A directory tree and its structure index.

the appropriate DAG index for each of the metadata conditions.

3.3 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 Equation 6. 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 notably, the *PathStack* algorithm [7] 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 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. 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 2).

We extend our index to maintain the position that each component appears within each directory (Figure 2). For efficiency, each directory is represented by an integer directory id, so that each entry in the index is a pair of integers ($DirID$, $position$).[2] 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 [17].

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. 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

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

Algorithm 1 matchDirectory(Q)

1. termSet $\Leftarrow$ extractQueryTerms(Q)
2. dirIDPositionListMap $\Leftarrow$ { term : getDirIDPositionList(term) | term $\in$ termSet }
   {Given a term, *getDirIDPositionList* returns a list of (DirID,position) tuples stored in the inverted index.}
3. fileMatches $\Leftarrow$ {}
4. **for** each dirID that appears in all lists of dirIDPositionListMap **do**
5. positionMap $\Leftarrow$ { term : getPositionList(dirIDPositionListMap, term, dirID) | term $\in$ termSet }
   {Given term and dirID, *getPositionList* returns the list of positions stored in dirIDPositionListMap.}
6. **if** satisfyStructRelationships(positionMap, Q, 0, Q) **then**
7. matchedFiles $\Leftarrow$ matchedFiles $\cup$ getFiles(dirID)
   {Given dirID, *getFiles* returns all files under the directory represented by dirID.}
8. **return** matchedFiles

9. **function** satisfyStructRelationships(positionMap, remainingQ, position, Q)
10. **begin**
11. **if** remainingQ is empty **then**
12. **return** true
13. head $\Leftarrow$ leftMostEntity(remainingQ)
    {*leftMostEntity* returns the left-most query node (group) of a path query.}
14. **if** isNodeGroup(head) **then**
15. headPositions $\Leftarrow$ getNGPositions(positionMap, head, position)
16. **else**
17. headPositions $\Leftarrow$ getPositions(positionMap, head)
    {*getPositions* returns the positions of a given term that is stored in a term-position lookup table.}
18. **for** each pos $\in$ headPositions s.t. pos $>$ position **do**
19. **if** (isRootNode(head) **or** pos and the position of left adjacent node of head satisfy the PC/AD relationship in Q) **and** satisfyStructRelationships(positionMap, remainingQ-head, pos, Q) **then**
20. **return** true
21. **return** false

Algorithm 2 getNGPositions(positionMap, NG, pos)

1. positions $\Leftarrow$ {}
2. tryNGPosition({}, NG, pos, NG, positions)
3. **return** positions

4. **function** tryNGPosition(selections, remainingNG, pos, NG, positions)
5. **begin**
6. **if** remainingNG is empty **then**
7. sortedSelection $\Leftarrow$ sort(selections)
8. **if** sortedSelection satisfies the sequence of PC/AD relationships in NG **then**
9. positions $\Leftarrow$ positions $\cup$ (min(sortedSelection), max(sortedSelection))
10. **return**
11. term $\Leftarrow$ nextUniqueTerm(remainingNG)
12. count $\Leftarrow$ numberOfOccurrence(remainingNG, term)
13. **for** each possible sel $\subseteq$ getPositions(positionMap, term) s.t. |sel| = count, sel $>$ pos **do**
14. tryNGPosition(selections $\cup$ sel, remainingNG-term, pos, NG, positions)

parts of the directory that match the node groups. Finally, we would attempt to find an ordering of components and node groups that would match the generated sequence of conditions. If we can find such an ordering, then the directory matches the query path; otherwise, it does not.

Our two-phase path matching approach is detailed in the *matchDirectory* algorithm shown in Algorithm 1. The algorithm first locates all inverted lists for the query terms (line 1-2). It then finds all candidate $dirID$s that appear in all inverted lists (line 4) and calls function $satisfyStructRelationships$ to check whether the positions of terms constitute a match for the query (line 6). The algorithm collects all $dirID$s whose pathnames match the query and returns the files under these directories (line 7-8).

In function satisfyStructRelationships, from left to right the algorithm finds all occurrences of each query node (group) (line 14-17) and checks whether the node (group) satisfies the parent-child or ancestor-descendent relationship for its adjacent node (group) (line 19). The function returns true as soon as a match is found (line 12, 20).

Algorithm 1 calls function $getNGPositions$ and $tryNGPosition$ (Algorithm 2) to obtain all occurrences of a node group in a directory pathname. The function *tryNGPosition* enumerates all combination of the positions of unique terms (line 11-14) and checks whether they match the sequence of parent-child or ancestor-

descendent relationships of the node group (line 7-8). All matches are then collected and returned from the functions (line 3, 9).

Note that we use a pair of positions $(position_{head}, position_{tail}) = (\min(sortedSelection), \max(sortedSelection))$ (line 9) to represent the occurrence of a match for a node (group). For a node, $position_{head}$ equals $position_{tail}$. For a node group, $position_{head}$ and $position_{tail}$ represent the highest and lowest position of a match respectively. When checking the structural relationships, we use $position_{head}$ and $position_{tail}$ to check the parent-child or ancestor-descendant relationship with the left and right adjacent node (group) respectively.

As an example, 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 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.

For a path query Q with $|Q|$ components, our query matching algorithm has worst-case I/O complexities linear in the sum of sizes of the $|Q|$ inverted lists. And the worst-case space complexity does not exceed the maximum length of directory pathnames[3].

The worst-case CPU time complexity of our query matching algorithm is $L_{min} \cdot C$, where L_{min} is the length of the shortest inverted list for all query components, and C is the time to match a directory pathname against Q. This is because in the worst case the algorithm needs to check at most L_{min} directory pathnames due to the andish condition for query components. For a directory pathname P with $|P|$ components, since the algorithm enumerates all possible combination of positions of terms and checks whether they satisfy the structural relationships, in the worst case the computation complexity of C is $O(|P|^{|Q|} \cdot |Q| \cdot \log(|Q|))$. However, in practice this is not a big issue because:

- Often the number of terms of a pathname that appear in a query is much smaller than $|P|$. And often there are few duplicated terms in a query.
- The query length $|Q|$ is usually a small number, which can be treated as a small constant.
- For our data set, in most of the cases the path matching algorithm terminates quickly either because a match is found and no further computation is necessary, or a term has few occurrences in a directory pathname and no-match is detected quickly.

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.

Given the above matching algorithm, we can then support TA in the structure dimension by dynamically building the DAG and populating its nodes with score information. (Building a static structural index is not a realistic option as this would entail enumerating all possible query conditions (paths) and all of their relaxations, a prohibitively expensive task.) A naive implementation of sorted access could then be a DAG traversal in decreasing order of structure scores. Similarly, random access could be implemented as a DAG traversal to locate the least relaxed query that a file matches. However, complete expansion and scoring of the DAG would be too expensive. Thus, in the next section, we present optimizations to minimize the expansion and scoring of the DAG.

4 Optimizing Query Processing in the Structure Dimension

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

[3]Function $getNGPositions$ may compute all occurrences of a node group in a lazy fashion. It only tries to find the next match upon a new request. In this way, it does not have to store all occurrences at once.

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 [2, 20] 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 3). 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 4.1). 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). 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 4.3).

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

4.1 Incremental Identification of Relaxed Matches

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. The DAG

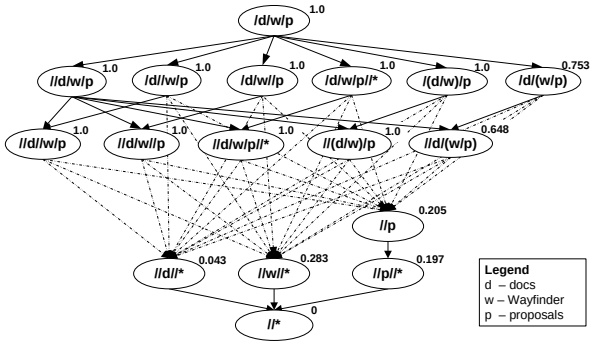


Figure 3: Structure relaxation DAG. Solid lines represent parent-child relationships. Dotted lines represent ancestor-descendant relationships, with intermediate nodes removed for simplicity of presentation. IDF scores are shown at the top right corner of each DAG node.

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 [2, 20]. An example DAG is given in Figure 3 for the query condition */docs/Wayfinder/proposals*. Only one exact match for the query condition exists so the score of 1 is associated with the DAG root node. As we go down the DAG, the increasingly relaxed versions of the query condition match more and more files resulting in lower *IDF* scores. The most relaxed version of the query condition: //* matches all files and it has a score of 0.

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 in a greedy fashion. The partial evaluation should nevertheless ensures that directories (and therefore files) are

Algorithm 3 IncrementalDAG($root$, k)

```
1.  Queue ⇐ PriorityQueue(getScore(root), root)
2.  topKFiles ⇐ {}
3.  currentNode ⇐ nil
4.  seenNodes ⇐ {root}
5.  while notEmpty(Queue) do
6.    if currentNode ≠ nil then
7.      childNodes ⇐ 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 ∈ childNodes do
9.        if n ∉ seenNodes then
10.         addNode(Queue, getScore(n), n)
11.         seenNodes ⇐ seenNodes ∪ {n}
12.   currentNode ⇐ removeNode(Queue)
13.   fileMatches ⇐ matchDirectory(getQuery(currentNode))
      {matchDirectory is described in detail in Section 3.3.}
14.   topKFiles ⇐ topKFiles ∪ fileMatches
15.   if |topKFiles| ≥ k then
16.     return topKFiles
17. return topKFiles
```

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 described in detail in [20]. 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.[4] In the next sections we will discuss techniques to optimize sorted and random access to the query relaxation DAG.

4.2 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.

An interesting observation is that not every possible relaxation leads to the discovery of new matches. For example, in Figure 3, 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* (Equation 6). 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 1

Theorem 1 *Given the structural $score_{idf}$ function defined in Equation 6, 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 as $score_{idf}(P)$, and $F(P') = F(P'') = F(P)$, where $F(P)$ is the set of files matching query path P.*

Proof. (Sketch) If $score_{idf}(P') = score_{idf}(P)$, then by definition $N_{P'} = N_P$ (Equation 6). 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

[4] 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 4 DAGJump(srcNode)

```
1.  s ⇐ getScore(srcNode)
2.  currentNode ⇐ srcNode
3.  loop
4.     targetDepth ⇐ getDepth(currentNode)
5.     childNode ⇐ firstChild(currentNode)
6.     if getScore(childNode) ≠ s or
         hasNoChildNodes(childNode) then
7.        exit loop
8.     currentNode ⇐ 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, ∀m ∈ S
11.    for each m ∈ S do
12.       for each n' on path p ∈ getPaths(n, m) do
13.          setScore(n', s)
14.          setSkippable(n')
15.       if notSkippable(m) then
16.          setSkippable(m)
```

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 1 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 a previously scored node P^*, where $score_{idf}(P^*) = score_{idf}(P)$[5]; 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. 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.

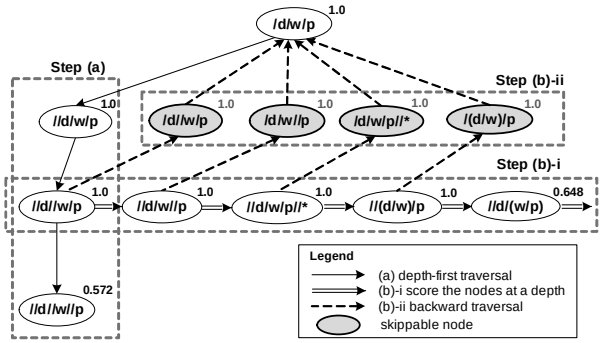


Figure 4: An example execution of *DAGJump*. IDF scores are shown at the top right corner of each DAG node.

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.

4.3 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

[5]This condition is guaranteed to occur because of our *IncrementalDAG* algorithm.

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)

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, 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 5 which shows the

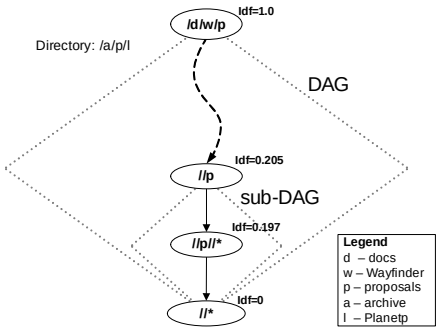


Figure 5: An example execution of *RandomDAG* that returns the score of a file that is in the directory */archive/proposals/Planetp* for the query condition */docs/Wayfinder/proposals*.

parts of the DAG from Figure 3 that would need to be accessed for a random access to the score of a file that is in the directory */archive/proposals/Planetp*.

The following theorems are the key to establishing the correctness of *RandomDAG* Algorithm.

Theorem 2 *Let P (containing keyword set S_1) be a full pathname. Then, in a structure DAG rooted at a path query Q (containing keyword set S_2), a query Q' (other than the match-all query //*) matched by P must contain and only contains keywords from the intersection of two sets $S_1 \cap S_2$. Here we exclude match-all node $*$ from the keyword set.*

Proof. (Sketch) We prove it by contradiction. Assume the conclusion is not true so a path query Q' (containing keyword set S_3) other than $//*$ and matched by P has at least one keyword w s.t. $w \in S_3$ and $w \notin S_1 \cap S_2$. Based on the definition of path query, any full path matching Q' must contain keyword w so $w \in S_1$.

Since Q' is in the DAG rooted at Q, Q' is a relaxed version of Q. Because none of structure relaxations introduces new keywords, Q' only contains keywords from S_2. We have $w \in S_3 \subseteq S_2$. Because $w \notin S_1 \cap S_2$, it must be true $w \notin S_1$, contradicting our assumption $w \in S_1$. ■

Theorem 3 *In a structure DAG rooted at a path query Q (containing keyword set S_1), among all queries containing keywords only from a keyword set S_2, there exists a*

least relaxed query Q' obtained by applying to Q a series of node deletions for keywords from $S_1 - S_2$.

Proof. (Sketch) First, when applying a series of node deletion operations to a path query, we always get the same relaxed path query regardless of the order of node deletion operations. This is because changes made by node deletion are locally constrained within the adjacent nodes, node groups, and edges.

Second, if a node deletion follows a series of non-node-deletion operations, the result query is a relaxation of the query obtained by applying node deletion to the original query. This can be derived from the containment condition and the node deletion definition. By induction we can further prove this is also true when there are multiple node deletions and some non-node-deletion operations occur between node deletions.

Last, in the structure DAG any query Q'' that only contains keywords from S_2 is obtained by applying a series of relaxations (corresponding to a path from the root of DAG to a DAG node representing Q''). These relaxation operations can be divided into a set of node deletion operations, noted O_{ND}, and a set of non-node-deletion operations, noted $O_{\neg ND}$. According to the above conclusions, Q'' is a relaxed version of a query Q' that is obtained by applying node deletion operations in O_{ND} to query Q, regardless of the order of these operations. Therefore, Q' is the least relaxed query among the queries that only contain words from S_2, and O_{ND} is the set of node deletions where each deletes a keyword in $S_1 - S_2$. ■

Based on these two theorems, given a file F and a DAG, the query Q'' matched by F only contains keywords in common from the pathname p of F and the root node query Q of the DAG. Furthermore, Q'' must be the descendant node of a DAG node obtained by applying node deletion operations to Q for the keywords in $p - Q$. This proves the correctness of Algorithm 5.

5 Experimental Results

We now turn to evaluating the search performance achievable using the indexing structures, scoring algorithms, and top-k algorithm described in Sections 3 and 4. 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 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.

5.1 Experimental Setup

Experimental environment. We evaluate the performance of our search approach by measuring the query processing time of a prototype that we implemented in the Wayfinder file system. Our prototype and Wayfinder are completely written in Java. We use the Berkeley DB [21] to persistently store all indexes across restarts of the file system. All 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. 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. As noted in [10], there is a lack of synthetic data sets and benchmarks to evaluate search over personal information management systems. Thus, we use a data set that contains files and directories from the working environment of one of the authors. This data set contains 14.3 GB of data from 24,926 files organized in 2,338 directories; 24% of the files are multi-media files (e.g., music and pictures), 17% document files (e.g., pdf, text, and MS Office), 14% email messages,[6] and 12% source code files. The average directory depth was 3.4 with the longest being 9. On average, each directory contains 11.6 subdirectories and files, with the largest—a folder containing emails—containing 1013. Wayfinder extracted 347,448 unique stemmed content terms.[7] File modification dates span 10 years. 75% of the files are smaller than 177 KB, and 95% of the files are smaller than 4.1 MB.

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

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

Query Evaluation Results						
	Query Conditions					Comments on Relaxation
Query	Content	Type	Modification Date	Structure	Rank	from Query Q1/Q14
Search Scenario 1: The user searches for the electronic book "The Time Machine". **Target file:** /Personal/Ebooks/Novels/The Time Machine.pdf **Structure condition:** /Personal/Ebooks/Novels (abbreviated as /p/e/n, 'c' in the incorrect path stands for 'Comics')						
Q1	time, machine	-	-	-	18	Base Query
Q2	time, machine	.pdf	22 Jan 07 18:09	$/p/e/n$	1	Correct Values (all dimensions)
Q3	time, machine	.pdf	-	-	9	Correct File Type
Q4	time, machine	.doc	-	-	42 *	Incorrect File Type
Q5	time, machine	Docs.	-	-	18	Relaxed Range
Q6	time, machine	-	21-27 Jan 07	-	1	Relaxed Range (Week of month)
Q7	time, machine	-	23 Jan 07 18:09	-	1 *	Incorrect Date (off by 1 day)
Q8	time, machine	-	15 Feb 07 18:09	-	4 *	Incorrect Date (off by 1 month)
Q9	time, machine	-	-	$/p/e/n$	1	Correct Path
Q10	time, machine	-	-	$/n/e$	1 *	Incorrect Order/Correct Components
Q11	time, machine	-	-	$/p/e/c$	2 *	Incorrect Path
Q12	time, machine	Docs.	Jan 07	$/p/e$	1	Relaxed Range (all dimensions)
Q13	time, machine	.pdf	15 Feb 07 18:09	$/c/e$	1 *	Incorrect Date and Path
Search Scenario 2: The user searches for an email that discusses the java implementation of the IR algorithm for the file system Wayfinder. **Target file:** /Personal/Mail/Code/Java/920-SA____-20061018192157-78.xml **Structure condition:** /Mail/Code/Java (abbreviated as /m/c/j, 'p' in the incorrect path stands for 'Python')						
Q14	Wayfinder, IR	-	-	-	42	Base Query
Q15	Wayfinder, IR	.xml	18 Oct 06 14:21	$/m/c/j$	1	Correct Values (all dimensions)
Q16	Wayfinder, IR	.xml	-	-	35	Correct File Type
Q17	Wayfinder, IR	.txt	-	-	39 *	Incorrect File Type
Q18	Wayfinder, IR	Docs.	-	-	39	Relaxed Range
Q19	Wayfinder, IR	-	15-21 Oct 06	-	1	Relaxed Range (Week of month)
Q20	Wayfinder, IR	-	17 Oct 06 14:21	-	1 *	Incorrect Date (off by 1 day)
Q21	Wayfinder, IR	-	18 Nov 06 14:21	-	8 *	Incorrect Date (off by 1 month)
Q22	Wayfinder, IR	-	-	$/m/c/j$	1	Correct Path
Q23	Wayfinder, IR	-	-	$/j/m$	1 *	Incorrect Order/Correct Components
Q24	Wayfinder, IR	-	-	$/m/c/p$	1 *	Incorrect Path
Q25	Wayfinder, IR	Docs.	Oct 06	$/m/c$	1	Relaxed Range (all dimensions)
Q26	Wayfinder, IR	.txt	18 Nov 06 14:21	$/j/c$	1 *	Incorrect Date and Path

Table 1: The rank of target files returned by a set of related queries. The queried dimensions include *Content*, *Type* (Metadata), *Modification Date* (Metadata), and *Structure*.

5.2 Impact of Flexible Multi-Dimensional Search

We begin by exploring the potential of our approach to improve scoring (and so ranking) accuracy using two example search scenarios. In each scenario, we initially construct a content-only query intended to retrieve a specific target file and then expand this query along several other dimensions. 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.

A representative sample of results is given in Table 1. In the first example, the target file is the novel "The Time Machine" by H. G. Wells, located in the directory path */Personal/Ebooks/Novels/*, and the set of query content terms in our initial content-only query Q1 contains the two terms *time* and *machine*. While the query is quite reasonable, the terms are generic enough that they appear in many files, leading to a ranking of 18 for the target file. Query Q2 augments Q1 with the exact matching values for file type, modification date, and containing directory. This brings the rank of the target file to 1. The remaining queries explore what happens when we provide an incorrect value for the non-content dimensions. For example, in query Q10, a couple of correct but wrongly ordered components in the directory name still brings the ranking up to 1. 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; queries which contain a "*" next to our technique's rank result represent those in which the target file would not be considered as a relevant answer given today's typical filtering approach.

Results for the second example, which is a search for an email, are similar.

This study also presents an opportunity for gauging

the potential impact of the node inversion relaxation. Specifically, queries Q23 and Q26 in the second example misorder the structure conditions as */Java/Mail* and */Java/Code*, respectively, compared to the real pathname */Personal/Mail/Code/Java*. Node inversion allow these conditions to be relaxed to *//(Java//Mail)* and *//(Java//Code)*, so that the target file is still ranked 1. Without node inversion, these conditions cannot match the target until they both are relaxed to *//Java/**, the matching relaxation with the highest IDF score, using node deletion. This leads to ranks of 9 and 21 since files under other directories such as */Backup/CodeSnippet/Java* and */workspace/BookExample/Java* now have the same structure scores as the target file.

In another example scenario not shown here, a user is searching for the file *wayfinder_cons.ppt* stored in the directory */Personal/publications/wayfinder/presentations*. The query with content condition *wayfinder, availability, paper* and structure condition */Personal/wayfinder/presentations* would rank *wayfinder_cons.ppt* 1. However, if the structure condition is misordered as */Personal/presentations/wayfinder* or */presentations/Personal/wayfinder*, the rank of the target file would fall to 17 and 28, respectively, without node inversion. With node inversion, the conditions are relaxed to */Personal//(presentations/wayfinder)* and */(presentations//Personal/wayfinder)*, respectively, and the target file is still ranked 1.

5.3 Comparing with Existing Search Tools

We compare the accuracy of our multi-dimensional approach with TopX [25], a related approach designed for XML search, Google Desktop Search (GDS), and Lucene.

Query sets. We consider a set of 40 synthetically generated search scenarios similar to those considered in the last section. Specifically, 20 emails and 20 XML documents (e.g., ebooks) were randomly chosen to be search targets. Choosing XML documents (emails are stored in XML format) allows internal structure to be included in TopX queries. We then constructed 40 queries, one to search for each target file, for each search approach.

For our multi-dimensional approach, each query targeting a file f contains content, metadata, and structure conditions as follows:

- *Content:* 2 to 4 terms chosen randomly from f's content.
- *Metadata:* A date (last modified) 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:* a partial path p is formed by the correct ordering of 2 to 4 terms randomly chosen from f's parent directory pathname. The condition is then form by randomly choosing between: (a) using p, (b) dropping one random term from p, (c) mis-ordering a random pair of adjacent terms in p, and (d) misspelling one random term in p.

For GDS, each query contains the same content and structure terms as the multi-dimensional queries since GDS indexes both content and terms from directory pathnames. GDS do not differentiate between the two types of terms in the queries though, so each query is simply a set containing terms of both types.

For TopX, we choose content terms for each query as above. In addition, 1 to 2 XML tags that are parents of the chosen content terms are randomly chosen. Each query then contains the tags, content terms, and accurate structural relationships between all components.

For Lucene, we just use content terms for each query as above since Lucene only index textual content.

Note that this is not meant to be an "apple-to-apple" comparison since the different tools index different information, and so the queries are different. Also, we introduce inaccuracies in queries for our multi-dimensional approach but not the others. Thus, the study is only meant to assess the potential increase in search accuracy that comes with additional information of different types in the queries.

Figure 6 plots the cumulative distribution function (CDF) of ranks of target files returned by the different search approaches. Table 2 presents the average recall, precision, and mean reciprocal rank (MRR) for each

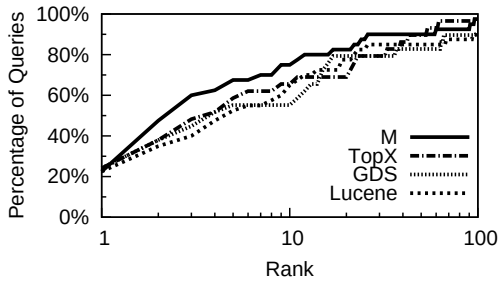


Figure 6: The CDFs of ranks of the target files for multi-dimensional search (M), TopX, GDS, and Lucene.

search technique for $k = 5, 10$. Recall and MRR are averages across the entire sets of 40 queries. Precision, however, is computed from a randomly selected subset of 6 queries for each technique. This is because computing precision is very time consuming, requiring the manual inspection of each top k files returned for each query and deciding whether the file is relevant to the query. The recall metric is not the standard IR recall because we do not have a relevance assessment for every file against every query. Instead, recall is the percentage of time that the target file is returned as one of the top k answers.

Observe that GDS mostly outperforms content-only search (Lucene) by using pathname terms. TopX mostly outperforms GDS and content-only search by using internal structure information. Finally, our approach performs the best by leveraging external structure information and metadata. We plan to integrate internal structure into our method in the future to further improve search accuracy.

5.4 Base Case Query Processing Performance

We now turn to evaluating the search performance of our system. 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) and RandomDAG (Section 4.3) optimization algorithms. Note that the base case still includes the implementation of the matchDirectory (Section 3.3) and incremental DAG building (Section 4.1) techniques.

Query set. We expand the query set used in the last section for this study. Specifically, we add targets and queries for 40 additional search scenarios, 20 targeting additional (mostly non-XML) documents and 20 targeting media files (music, etc.). Queries are constructed in the same manner as before. 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.

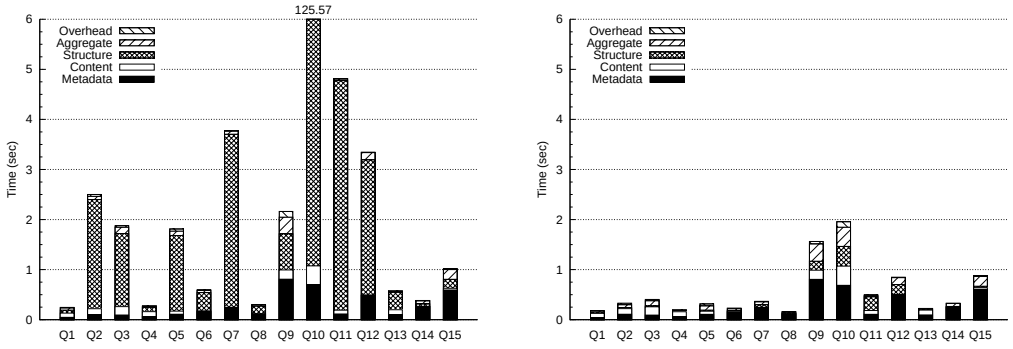
Choosing k. Query performance is a function of k, the number of top ranked results that should be returned to the user. We consider two factors in choosing a k value: (1) the mean recall (as defined above) and MRR, and (2) the likelihood that users would actually look through all k returned answers for the target file. We choose $k = 10$ as a reasonable trade-off between these two factors. For the set of 80 queries, recall/MRR are 0.85/0.61 for $k = 10$, which are somewhat smaller than the 0.95/0.62 for $k = 20$. Experience from Web search, however, shows 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 5.8.

Results. Figure 7(a) presents the query processing times, including breakdowns of the times 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 8 plots the 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 7. For the entire set of 80 queries, processing times in the structure dimension comprised at least 63% of the

Method	$k = 5$			$k = 10$		
	Recall	MRR	Precision	Recall	MRR	Precision
M	0.68	0.42	0.33	0.75	0.43	0.17
TopX	0.57	0.37	0.17	0.65	0.38	0.12
GDS	0.55	0.36	0.14	0.55	0.36	0.08
Lucene	0.53	0.35	0.13	0.65	0.36	0.08

Table 2: The mean recall and mean reciprocal rank (MRR) of the target file, and average precision for $k = 5, 10$.



(a) Base Case **(b) DAGJump+RandomDAG**

Figure 7: Breakdowns of query processing times for 15 queries for (a) the base case, and (b) DAGJump+RandomDAG case.

overall query processing times for the 21 slowest queries (those with processing times ≥ 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.

5.5 Query Processing Performance with Optimizations

Figure 7(b) gives the query processing times for the same 15 queries shown in Figure 7(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.

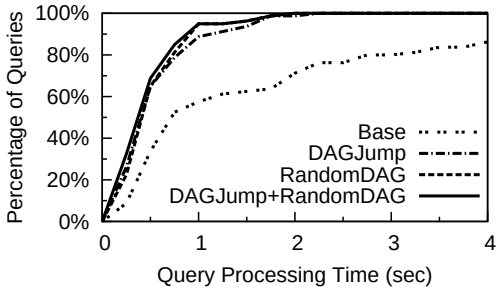


Figure 8: The CDFs of query processing time.

To view the overall effects of the optimizations on all queries, Figure 8 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,

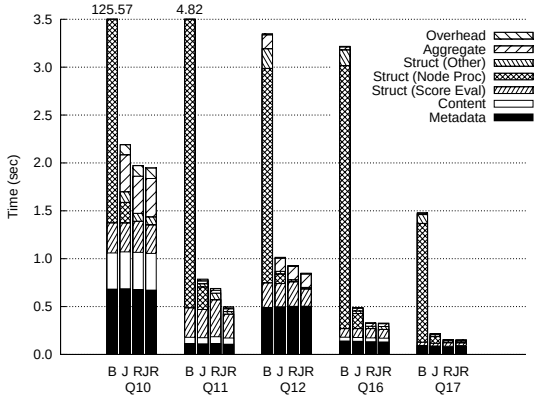


Figure 9: Query time breakdown for five representative queries with different impact from various optimizations. B, J, R, JR denote base, DAGJump, RandomDAG, and DAGJump+RandomDAG cases respectively.

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, 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.

5.6 Understanding the Impact of DAGJump and RandomDAG

Interestingly, just using RandomDAG alone achieves much of the benefits of using both optimizations together (Figure 8). 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 9 shows break downs of query processing times for five queries (three from Figure 7), 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.

5.7 Storage Cost

We report the cumulative size of our static indexes of Section 3 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. The indexes are so compact compared to the data set because

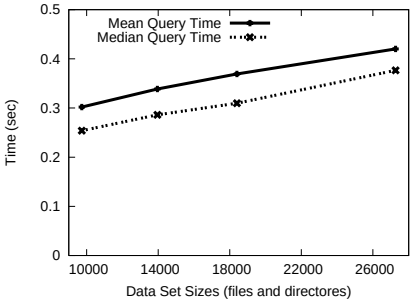


Figure 10: The mean and median query times for queries targeting email and documents plotted as a function of data set size.

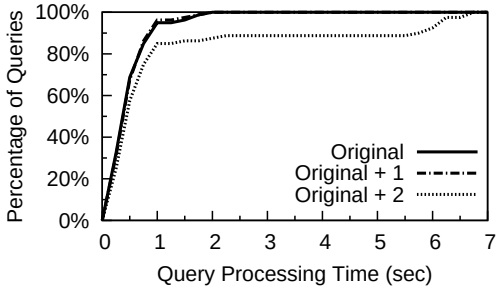


Figure 11: The CDFs of query times for one (Original), two (Original+1), and three (Original+2) structural query conditions.

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.

5.8 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 10, 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 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.

Finally, Figure 11 plots CDFs of query processing times when we extend each query studied in Section 5.5 to have 2 (Original + 1) and 3 structure conditions (Original + 2). Note that this means some queries contain 3 structure conditions, each of which can contain 4 components, possibly with an embedded inaccuracy (e.g., swap of adjacent components). We expect that in practice, few structure query conditions would reach this size. Yet, observe that there is no performance degradation when adding 1 structure condition. When adding 2 conditions, 85% of the queries still complete within 1 second. The tail of the CDF does stretch but the worse query time is still just 6.75 seconds.

6 Related Work

Several works have focused on the user perspective of personal information management [8, 16]. 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 [10, 28] address information management by proposing generic data models for heterogeneous and evolving information. These works are aimed at providing users with generic and flexible data models to accessing and storing information beyond what is supported in traditional files system. Instead, we focus on querying information that is already present in the file system. An interesting future direction would be to include entity as-

sociations in our search and scoring framework.

Other 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 [22] or by altering the model of the file system to a more object-orientated database system [6]. 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, 14]. These projects provide search capabilities over content and but use other query dimensions, such as size, date, or types, as filtering conditions. Research has shown that desktop users tend to prefer location-based search to a keyword-based search [5], which observes that users rely on the information such as directories, dates, and names when searching files. Another study [24] 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. These studies motivate the need for personal information management system search tools that use metadata and structure information.

The INitiative for the Evaluation of XML retrieval (INEX) [15] 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 [13] 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 [1, 3, 2]. Our work uses ideas from these previous works, such as the DAG indexing structure to represent all possible structural relaxations [2], and the relaxed query containment condition [3, 2]. We introduce node inversion, an important novel structure relaxation for personal information systems (as discussed in Section 5.6). This in turn led to the introduction of node groups, and necessitated algorithms for matching node groups. We also designed DAGJump and RandomDAG to optimize query processing. Finally, we include meta-data as an additional dimension that users can use to improve search accuracy.

TopX [25] is another related effort in XML retrieval. While TopX considers both structure and content, it is different from our work in a number of ways. Specifically, we consider external structure (directory pathnames) while TopX considers internal structure (i.e., XML). Furthermore, we use query relaxation to find and score approximate matches, whereas TopX transitively expands all structural dependencies and counts the number of conditions matched by a file to score structure. We plan to extend our work to consider internal structure in the future but our approach will still be based on query relaxation.

Several works such as [7, 17] have studied efficient path matching techniques for XML documents. In [17], path queries are decomposed into simple subexpressions and the results of evaluating these subexpressions are combined or joined to get the final result. In [7], a *PathStack* is constructed to track partial and total answers to a query path by iterating through document nodes in their position order. Our $DepthMatch$ algorithm considers node permutations, which are not easily supported by *PathStack*.

7 Conclusions

In this paper, 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 in-

dexes 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 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.

In the future, we plan to extend our scoring dimensions to include file context information, in the spirit of [22]. We also plan to explore the internal structure information for files and study the effectiveness of our algorithms with the added information.

References

[1] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree Pattern Relaxation. In *Proc. of the Intl. Conference on Extending Database Technology (EDBT)*, 2002.

[2] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman. Structure and Content Scoring for XML. In *Proc. of the Intl. Conference on Very Large Databases (VLDB)*, 2005.

[3] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. In *Proc. of the ACM Intl. Conference on Management of Data (SIGMOD)*, 2004.

[4] Lucene. `http://lucene.apache.org/`.

[5] D. Barreau and B. A. Nardi. Finding and Reminding: File Organization from the Desktop. *ACM SIGCHI Bulletin*, 27(3), 1995.

[6] C. M. Bowman, C. Dharap, M. Baruah, B. Camargo, and S. Potti. A File System for Information Management. In *Proc. of the Intl. Conference on Intelligent Information Management Systems (ISMM)*, 1994.

[7] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. of the ACM Intl. Conference on Management of Data (SIGMOD)*, 2002.

[8] Y. Cai, X. L. Dong, A. Halevy, J. M. Liu, and J. Madhavan. Personal Information Management with SEMEX. In *Proc. of the ACM Intl. 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 Intl. Conference on Research and Development in Information Retrieval (SIGIR)*, 2003.

[10] J.-P. Dittrich and M. A. Vaz Salles. iDM: A Unified and Versatile Data Model for Personal Dataspace Management. In *Proc. of the Intl. Conference on Very Large Databases (VLDB)*, 2006.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. *Journal of Computer and System Sciences*, 2003.

[12] M. Franklin, A. Halevy, and D. Maier. From Databases to Dataspaces: a New Abstraction for Information Management. *SIGMOD Record*, 34(4), 2005.

[13] N. Fuhr and K. Großjohann. XIRQL: An XML Query Language Based on Information Retrieval Concepts. *ACM Transactions on Information Systems (TOIS)*, 22(2), 2004.

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

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

[16] D. R. Karger, K. Bakshi, D. Huynh, D. Quan, and V. Sinha. Haystack: A General Purpose Information Management Tool for End Users of Semistructured Data. In *Proc. of the Conference on Innovative Data Systems Research (CIDR)*, 2005.

[17] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of the Intl. Conference on Very Large Databases (VLDB)*, 2001.

[18] C. Peery, F. M. Cuenca-Acuna, R. P. Martin, and T. D. Nguyen. Wayfinder: Navigating and Sharing Information in a Decentralized World. In *Databases, Information Systems, and Peer-to-Peer Computing - Second International Workshop, (DBISP2P)*, 2004.

[19] C. Peery, W. Wang, A. Marian, and T. D. Nguyen. Fuzzy Multi-dimensional Search in the Wayfinder File System. In *Proc. of the Intl. Conference on Data Engineering (ICDE)*, 2008.

[20] C. Peery, W. Wang, A. Marian, and T. D. Nguyen. Multi-Dimensional Search for Personal Information Management Systems. In *Proc. of the Intl. Conference on Extending Database Technology (EDBT)*, 2008.

[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] J. Teevan, C. Alvarado, M. Ackerman, and D. Karger. The Perfect Search Engine is Not Enough: A Study of Orienteering Behavior in Directed Search. In *Proc. of the Conference on Human Factors in Computing Systems (SIGCHI)*, 2004.

[25] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum. TopX: Efficient and Versatile Top-k Query Processing for Semistructured Data. *VLDB Journal*, 17(1), 2008.

[26] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Inc, 1999.

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

[28] 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)*, 2003.