Unified Structure and Content Search for Personal Information Management Systems

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

Technical Report DCS-TR-661
Department of Computer Science, Rutgers University
110 Frelinghuysen Rd, Piscataway, NJ 08854

December 14, 2009

Abstract

The amount of data that users are storing and accessing in personal information systems is growing massively. At the same time, the organization of this data is becoming more heterogeneous, with data spread across different organizational domains such as emails, music databases, and photo albums, some of which are structured by applications rather than users. Powerful search tools are needed to help users locate data in these rapidly expanding yet fragmented data sets. In this paper, we present a novel fuzzy search approach that considers approximate matches to structure and content query conditions. Our approach includes a scoring framework for computing unified relevance scores for potential answers. Critically, our framework uses unified data and query processing models so that structure conditions can be approximately matched by content inside files and vice versa. Our model also unifies external structure (directories) with internal structure (e.g., XML structure), allowing users to specify integrated queries that are matched to a single unified data domain. We propose indexes and algorithms for efficient query processing. Finally, we empirically evaluate our approach using a real data set. We show that our unified fuzzy search approach can leverage structure information to significantly improve search accuracy, yet is robust to mistakes in query conditions.

1 Introduction

The amount of data stored in personal information management systems is rapidly increasing, following the relentless growth in capacity and dropping price per byte of storage. This explosion of information is driving a critical need for search tools to retrieve 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 locate personal information stored in file systems, such as the commercial tools Google Desktop [15] and Spotlight [21]. 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) as a *filtering* condition. Recently, the research community has turned its focus on search over to Personal Information and Dataspaces [7, 11, 13], which consist of heterogeneous data collections. However, similar to the commercial 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 consider files as sets of words and do not exploit the rich structural information typically available in personal information systems. Unlike

searches over digital libraries and the Web, users searching their personal files possibly have in-depth knowledge of where they expect the file to be located (directory structure) and of structural information contained within the file (internal structure). Internal structure can be derived from the file format, e.g., `<from>` and `<to>` fields in email files, or could consist of user annotations, e.g., tags given to photo files. This additional information has the potential to significantly increase the accuracy of search in personal information systems. However, it is too rigid to use structural information only as filtering conditions since any mistake in the query will lead to relevant files being missed; a flexible approach allowing for some error in the structure conditions is desirable, as illustrated by the following example.

Example 1 *Consider John, a user saving personal information in the file system of a computing device. John wants to retrieve photos of a Halloween party that was held at his home where someone was wearing a witch costume.*

Ideally, the file directory structure would have been created and maintained consistently and all photos are properly tagged. In real-life scenarios, this is rarely the case: users change their file organizations over time, inconsistently annotate their data, and may gather information from different sources. In our example, John has changed the way he organizes his photos over time, and his pictures are not consistently tagged. As a result, pictures from Halloween parties held in different years match very different directory structures and do not necessarily have matching tags, as illustrated in Figure 1. In addition, some relevant pictures are not in John's main photo collection, but in his email folder as they were sent to him by friends.

This structural heterogeneity complicates the search for specific pictures. A content-only search for "Halloween, home, witch", even considering only pictures, is likely to result in many matches, of various relevance. None of the pictures –pic1728.gif, party42.jpg, and IMG_1391.gif– *contained in the example data set of Figure 1, contains all three keywords.* pic1728.gif *and* party42.jpg *contain two of the keywords; their relative rankings would depend on the underlying content scoring function.* IMG_1391.gif *is however arguably the best match as its directory structure contains the third missing keyword.*

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

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

Current search tools would probably return IMG_1391.gif *as an exact match to the query but would likely miss approximate but relevant matches for several reasons:*

- *By considering the structure part (`//home//caption`) as a filtering condition, image files that are very relevant to the content search part of the query, but which do not satisfy the exact structure condition would not be considered as valid answers. For example, the file* party42.jpg, *which contains the keywords "witch" and "home" but does not have a caption tag with value "Halloween" would not be returned although it may be a suitable approximate match to the query.*
- *Since the external (directory structure) and the internal (structure and content) are strictly separated in both the data and query models, answers that do not adhere to this strict separation would be missed. For example, the file* pic1728.gif *would be left out since "Halloween" is expected to be a content term and not part of the structure hierarchy.*

Because of the data heterogeneity of personal information systems, we believe it is critical to support approximate matches on both the content and structural components of queries and to allow for query conditions to be evaluated across file boundaries. For this purpose, we use a data model that unifies the external and internal structure in addition to content as a large XML tree, in the spirit of [11] to represent user data. We propose a query model that supports approximation in both the structure and content components of queries, and allows for structure components to be matched by content terms and vice versa. In addition, we propose a unified scoring framework that simultaneously considers relaxed query conditions on structure and content to provide a unified score.

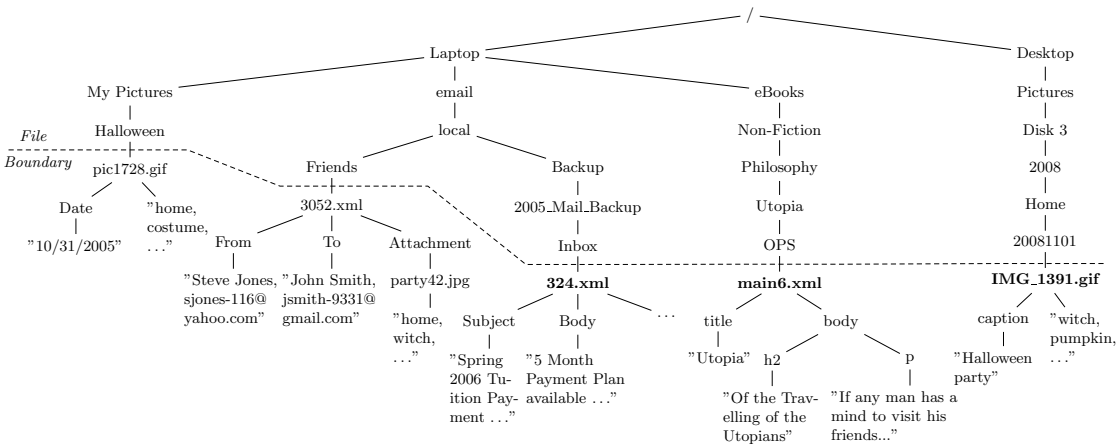


Figure 1: A subset of an example user personal information file system.

We make the following contributions:

- We propose unified data and query models to unify the structure outside and inside files and allow fuzzy matching of unified query conditions against both structure and content. Furthermore, matches in the unified data model may span multiple directories and files, giving users a rich query model for specifying contextual information in searches (Section 2).

- We develop a *TF·IDF*-based unified scoring framework to rank relevant search results (Section 3).

- We present query processing techniques to efficiently score answers. These techniques include indexes and a novel algorithm, *NIPathStack*, which is an extension of the popular *PathStack* algorithm[6] that handles matching of component permutations in queries (Section 4).

- We evaluate our models and query processing techniques and show that our unified approach is more accurate than filtering approaches and more robust than multi-dimensional approaches that consider structure and content separately (Section 5).

We discuss related work in Section 6 and conclude and discuss future work directions in Section 7.

2 Data and Query Model

2.1 Unified Data Model

Most users typically organize their files into a hierarchical directory structure for navigation. Fuzzy searches using this *external* structural information can significantly increase search accuracy [19]. In addition, the structure *within* a document can be seen as an extension of the directory structure and used to further improve searches in personal information management systems [11].

Our data model considers structure both outside and within files in a unified manner. Specifically, we model the entire file system as a rooted, labeled, unordered tree, that contains internal *structure* nodes and leaf *content* nodes. In the rest of the paper we refer to this data representation as the *unified data tree T*.

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

To simplify the discussion, we leave out file system metadata information such as file size or modification

time in this paper. Metadata can easily be included in the unified data tree, with both structural component (e.g., "Last Modified") and leaf node values (e.g., "10/31/09"). An interesting issue is whether metadata should be queried along with the rest of the data or separately; we plan to extend our query model to metadata in the future.

2.2 Flexible Query Model

Our model allows users to query both the content of files, using a standard keyword-based model, as well as their structure, internal and/or external. A query over our unified data model is a combination of structural components and content terms that can be represented as a twig, in the spirit of XQuery [24].

However, users typically cannot remember exactly where they stored a particular file or how the files are structured [10]. When structure information is part of a search query, the query is likely to be incomplete and may contain mistakes, as users often confuse or misremember the order of the structure components, their relationships, or their labels. The query structure is nevertheless useful as it gives meaningful context information to a general content query. Therefore it is desirable to provide an approximation mechanism that leverages any accurate structural information in the query without penalizing mistakes too strongly. Typical search systems that handle structural query information as filtering conditions often miss good results because of (possibly minor) user errors in the query. In contrast, our query model takes advantage of structural information, but allows for some flexibility in the way the data matches both structure and content conditions (see Section 3).

To represent this flexibility in our query model, we introduce the following notations for query nodes:

Root Node: The root node, noted $root$, is a query node that is matched by the root of the unified data tree *T*.

Content Node: A content node, noted *"N"*, is a query node that can be matched only by a leaf node in *T* with value N.

Structure Node: A structure node, noted *N*, is a query node that can be matched by any structure (non-leaf) node with label N in *T*.

Generalized Node: A generalized node, noted $\{N\}$, is a query node that can be matched by either a structure or content node with label or value N in *T*.

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

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

Node Group: To represent possible permutations of query nodes, we use node groups, as introduced in [19]. A node group is a path segment where all nodes are structure or generalized nodes, and all edges in the path are ancestor-descendant edges. Each node group may contain at most one generalized node (since generalized nodes can be matched by content nodes and each path contains at most one content node). The placement of the generalized node is fixed at the end of the path although the node labels may permute. Essentially, a node group *NG* is a query node that can be matched by all paths in *T* where each path matches a path segment *PS* that is a valid permutation of *NG*'s components.

Our model considers twig query patterns over the unified data tree *T*. A twig query is a tree pattern that starts at the root node of *T* ($root$), possibly containing multiple branches. Branches may end with any node type defined above. Generalized and content nodes can only be positioned as the last node of a branch.

In this paper, we consider a simplification of the query model that decomposes a twig query into a set of path queries for scoring, since it is complicated to allow flexibility such as component permutations (Section 3.1) for twig queries. We plan to lift this constraint and support flexible non-decomposed twig queries in the future. In our simplified model, a path query is created for each root to leaf path in the twig query.

We use path queries as the scoring units and compute the score of a twig query as a function of the scores of the

path queries resulting from the twig query decomposition (Section 3.2).

A *match* for a path query *PQ* in a data tree *T* is defined as a set of nodes in *T* such that there exists a mapping from query nodes of *PQ* to data nodes that preserves labels and structural relationships.

A potential answer to a path query is any path in our unified data tree that matches some relaxed form of the path query. Similar to many popular search approaches, we focus on a ranked query model where only the k best matches are returned to the user. The score of a match depends on the "closeness" of the match, as defined by a scoring function (see Section 3). While our model supports all possible granularity of query result (file, group of files, subtree within a file), for simplicity our current implementation only considers individual files as potential answers. We are planning to remove this restriction in future work.

We call the lowest matching node in a path that matches a path query a *match point*; each matching path has a unique match point. A file is an answer if its structure (including its full pathname and internal structure) and content contain one or more match points.

3 Scoring Framework

We now present our unified scoring framework. As already mentioned, our framework allows for approximation in both the structure and content dimension, as well as across the two dimensions. We score individual paths of a given query twig in a *TF·IDF*-based fashion; individual path scores are then combined together to produce a unified score.

3.1 Query Relaxations

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

We extend prior work on XML structural query relaxations [3, 19] to our unified query model. We use several types of structural relaxations, some of which were not considered in [3], to handle the specific needs of user searches in a file system. In addition, we augment the relaxations defined in [19] with relaxations that mix content and structure conditions and take into consideration the structure within a file to handle unified content and structure queries. As in [3, 19], we require that answers to a path query *P* be contained in the set of answers to any relaxation of *P* to ensure monotonicity of *IDF* scores (since *IDF* scores depend on the number of files that are answers to the path query).

We consider the following structural relaxation operations:

Edge Generalization is used to relax a parent-child relationship to an ancestor-descendant relationship.

Path Extension is used to extend a path query P to *P*//* so that any path containing P as a prefix will become a match. The rightmost edge of *P* must be an ancestor-descendant edge before path extension can be applied.

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

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

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

new query *P'*.

- If *n* is an internal node, *n* is dropped from *P*, and the nodes before and after n are connected in *P* with $//$.

To delete a node *n* that is within a node group *NG* in a path query *P*, the following steps are required to ensure answer containment:

- *n* and one of its adjacent edge in *N* are dropped from *NG*. If only one node *N* is left in *N*, *NG* is replaced by *N* in *P*.

- If *N* is a leaf node group, the result query is extended.

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 *//**, which matches all files in the unified data tree.

3.2 Scoring Methodology

Our scoring methodology is based on *TF·IDF* measures, as introduced in [2, 19]. Unlike these previous works, which compute separate scores for the content and structure dimensions and then aggregate them into a single score, the novelty of our approach is that both content and structure are scored together and our scoring framework allows for approximation within and across both dimensions. Our scoring functions are as follows.

Definition 1 (IDF Score of a Path Query) *Given a unified data tree T and a path query PQ, we define*

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

where $matches(T, PQ)$ is the set of all files in T that match PQ, and N is the total number of files in T.

Our *IDF* scoring formula guarantees that more relaxed forms of a query will receive lower scores since they are matched by more files, based on the containment property of relaxations.

Definition 2 (TF Score of a File for a given Query) *Given a path query PQ and a file F, we define*

$$score_{tf}(PQ, F) = f\left(\frac{F_{PQ}^{(struct)}}{|F^{(struct)}|}\right) + f\left(\frac{F_{PQ}^{(cont)}}{|F^{(cont)}|}\right)$$

where $|F^{(struct)}|$ and $|F^{(cont)}|$ are the number of structure and content nodes in F respectively, $F_{PQ}^{(struct)}$ and $F_{PQ}^{(cont)}$ are the number of structure and content match points in F that match PQ, and $f(x)$ is a function that affects the distribution of the score values, and so controls the relative impact of TF *on the overall score.*

While not shown in this paper because of space constraints, we have studied the impact of using different functions for $f(x)$ from the set $\{\log(1+x)\} \cup \{x^{\frac{1}{n}} | n = 2, 3, \ldots\}$, which are common alternatives for *TF* from the IR community, e.g., [18, 4]. For the data and query sets used in our evaluation (Section 5), $f(x) = x^{\frac{1}{10}}$ gives the best results.

Definition 3 (Unified Score of a File for a given Query) *Given a unified data tree T, a path query PQ, and a file F, we define*

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

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

Note that the overall score of a file is a summation across all possible relaxations of the query. As shall be seen, to achieve reasonable performance, our implementation is actually an approximation of the above equation. Abstractly, however, it is important to consider all relaxed forms of the query because it is difficult to determine the single best matching relaxed form for a given file. As the query becomes more relaxed, the *IDF* score is guaranteed to decrease (as already mentioned). However, *TF* may increase significantly because a more relaxed query may have many more match points in the file. Thus, the *TF·IDF* score of a file may be higher for a more relaxed form of the query. The summation provides an overall picture of how closely a file matches a given query by counting matching contributions across all possible relaxed forms of the query.

Unfortunately, it could be prohibitively expensive to compute $score(PQ, F)$ as defined in Definition 3 because the number of relaxed forms of a query grows exponentially with the query size. Thus, our implementation instead uses a lexicographical ordering, given by $(score_{idf}(LRQ_F^{PQ}), score_{tf}(LRQ_F^{PQ}, F))$, as an approximation, where LRQ_F^{PQ} is the least relaxed query that a file *F* matches. Our evaluation (Section 5) is entirely based on this lexicographical ordering because it is a tight approximation of the original scoring function (using $f(x) = x^{\frac{1}{10}}$ for *TF*)[1].

Finally, we compute the score of a file for a Twig Query *TQ* by computing the summation of scores of the file for each unique path query *PQ* derivable from *TQ*.

4 Query Evaluation

We now describe our query evaluation techniques, based on a top-k or ranked query processing model, which returns the best k answers for each query.

4.1 Index Structures

We index our unified data tree using an inverted index similar to those used in the XML community. In addition, because the possible query relaxations (Section 3.1) are query dependent, we need to build indexing structures to evaluate relaxations at runtime.

Indexing Data: We use an inverted index to enable fast access to the (potentially very large) unified data tree at query processing time. We assign a tuple of attributes *(FileId, PreCode, PostCode, Depth)* to each node of the unified data tree, where *FileId* is a unique identifier of the file containing *N* if *N* is part of a file, or 0 if *N* is a directory, *PreCode* and *PostCode* are values generated by a preorder and postorder traversal of the tree respectively, and *Depth* is the distance from root to *N*. The *PreCode*, *PostCode*, and *Depth* information are used to quickly determine the structural relationships (ancestor-descendant and parent-child) and are widely used in XML query processing [16]. The *FileId* is used to quickly identify answers (files) that match a particular query. The inverted index then maintains mappings from node labels and values–recall that each internal node has a label and each leaf has a value–to the nodes' attribute tuples.

Indexing Query Relaxations: We represent all possible relaxations of a query, along with the corresponding *IDF* scores for (files that match) each relaxation, using a DAG structure, as was proposed in [2, 19]. This DAG is created by incrementally applying query relaxations to the original query condition. By design, children of a DAG node are more relaxed versions of the query and therefore match at least as many answers as their parent (containment property). The most relaxed version of any query condition is the node *//**, which matches all files and so gives a *TF·IDF* score of 0.

Previous work [2, 19, 22] have detailed algorithms to efficiently build such a DAG structure. We adapt these algorithms to handle the full set of relaxations described in Section 3.1. Specifically, we use *IncrementalDAG* [22] algorithm to incrementally build a DAG and dynamically evaluate the structural relaxations.

4.2 Query Matching

Given a query, we need algorithms to efficiently use the indexing structures just described to identify and score matching answers. Several efficient algorithms have been proposed for XML pattern matching, one of the most popular being the *PathStack* algorithm [6]. *PathStack* views the XML data tree as a stream of nodes produced by a preorder traversal. The algorithm associates a stack S_N with each query node *N*, keeping the stack in the same order as the query nodes. It then pushes matching data nodes from the stream onto the stacks. Whenever a node is pushed onto the last stack, each unique sequence of nodes across all the stacks, one per stack, that satisfies the structural relationships in the query is an answer to the query. Nodes are popped from the stacks when processing moves to a different tree branch.

We cannot use *PathStack* directly because our query model includes node permutations in the form of node groups (Section 2.2). However, we have been able to adapt *PathStack* as follows:

1. For each node group *NG* in a query *P* we use a new *NIPathStack* algorithm to find all matches in

[1] Although not shown here due to lack of space, we empirically show the tightness of approximation for the data and query sets used in our evaluation (Section 5).

the unified data tree. These matches are returned in increasing lexigraphical ordering given by the pair ($PreCode_t$,$PreCode_h$).

2. We apply *PathStack* on *P* with a small variation: each node group *NG* is given an individual stack, and its matches are populated by the results of running *NIPathStack* on that node group. Other nodes are given a stack each as in the original *PathStack* algorithm.

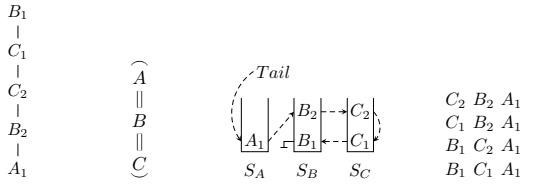
Like *PathStack*, *NIPathStack* uses a set of stacks, one per query node in a given node group, to find matches in our data tree and views the data tree as a stream of nodes produced by a preorder traversal. However, we use our inverted index (Section 4.1) to avoid traversing the entire tree. Specifically, each query node is associated with the inverted list keyed by the node's label or value. We then simulate the preorder traversal by considering the nodes from the matching inverted lists in sorted order according to their *PreCode*. As nodes are pushed onto the stacks, NIPathStack maintain pointers between these nodes to represent structural ancestor-descendant relationships in the data. As each new node is pushed onto one of the stack, the algorithm checks for solutions. At least one solution exists if all stacks are populated since node groups allow for any ordering of nodes within answers. When the traversal passes the leaf data node of a branch, nodes that cannot be involved in any new matches are popped from the stacks and the traversal moves to the next branch in the unified data tree.

Algorithm 1 details the *NIPathStack* algorithm. *NIPathStack* takes as input the node group being evaluated, noted *NG*. It keeps a pointer *Tail* to track the deepest node of the data path encoded in the stacks. Line 3 identifies the next node to be processed to simulate the preorder traversal, i.e., the inverted list which contains the node with the minimum *PreCode*. Line 4-6 pop stack nodes if the algorithm moves to the next branch during a preorder traversal of the unified data tree. The function *prevDataPathNode* is used to return the previous node in the data path encoded in stacks. Line 7-8 augment the data path with the new data node and assign the new deepest node of the data path. If at least one solution exists (function *containSolution* checks if all stacks are populated), line 10 invokes a sub-algorithm *NIShowSolutions* (Algorithm 2) to find and return all solutions.

Algorithm 1 $NIPathStack(NG)$

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

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



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

Figure 2: Data path and answer encoding in stacks.

Figure 2c shows the stack encoding of the data path

Algorithm 2 $NIShowSolutions(\textit{Tail}, NG)$

```
1.  P ⇐ (Tail) {partial answer with one node}
2.  showSolution(prevDataPathNode(Tail), P, NG)
3.  function showSolution(n, P, NG)
4.  begin
5.    if P does not contain a node with same label as n then
6.      P ⇐ (P, n)
7.      if |P| = |NG| then
8.        output(P)
9.      else
10.       showSolution(prevDataPathNode(n), P, NG)
11.     P ⇐ P − n
12.     if S_n has at least one node below n then
13.       showSolution(prevDataPathNode(n), P, NG)
14.   else
15.     if |P| < |NG| ∧ prevDataPathNode(n) ≠ nil
        then
16.       showSolution(prevDataPathNode(n), P, NG)
```

$B_1/C_1/C_2$ $/B_2/A_1$ (Figure 2a) for the node group *(A//B//C)* (Figure 2b). In this example, all data nodes are part of the same path and are therefore linked together. *NISHowSolutions* outputs answers in a leaf-to-root order, starting with the deep most node (here A_1), all answers ending with this node are then produced recursively, in our example: $B_1/C_1/A_1$, $B_1/C_2/A_1$. $C_1/B_2/A_1$, and $C_2/B_2/A_1$. Because the node group semantic allows for any ordering of nodes in data paths, the algorithm is guaranteed to return at least one path match if each stack contains at least one data node.[2]

4.3 Analysis of NIPathStack

Lemma 1 *Suppose that t_N is an arbitrary data node in the path (the chain of data nodes) encoded in stacks and we have that $getMinSource(n) = N'$. And suppose that $t_{N'}$ is the next element in N''s stream. Then, after $t_{N'}$ is pushed onto stack $S_{N'}$, the chain of data nodes from t_N to $t_{N'}$ verifies that their labels are included in a path in the unified structure from $t_{N'}$ to the root.*

For each node $t_{N_{min}}$ pushed onto stack $S_{N_{min}}$, it is easy to see that Lemma 1 holds given the Proposition 3.1 in [6].

Lemma 2 *The chain of the data nodes encoded in stacks contains at least one answer to the node group pattern n iff $\forall N_i \in n : |S_{N_i}| \geq 1$.*

It is also easy to see that for any query node N_i in n, if $|S_{N_i}| \geq 1$, at least one data node in the chain matches N_i. Because n only contains ancestor-descendant edges and we allow query nodes permute within n, according to Lemma 1, the chain at least contains one path that matches n.

Note the iterative nature of Algorithm 2 ensures that all possible answers that end with $t_{N_{min}}$ will be output. This leads to the correctness of our algorithms.

Theorem 4 *Given a node group pattern n and a unified structure T, Algorithm NIPathStack correctly returns all answers to n in T.*

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

Note *NIShowSolutions* is called only if the chain in stacks contains at least one answer. And unless an answer can be constructed from the rest of the chain, Line 8, 15, and 18 in Algorithm 2 prevent recursive invocation of *showNext*. Therefore, the cost to output each answer is bounded by the maximum length of a root-to-leaf path in the unified structure and the cost of *NIShowSolutions* is proportional to the size of the output list. And we have the following theorem.

Theorem 5 *Given a node group n and a unified structure T, Algorithm NIPathStack has worst-case I/O and CPU time complexities linear in the sum of sizes of the $|n|$ input lists and the output list. And the worst-case space complexity of Algorithm NIPathStack is the minimum of (a) the sum of sizes of the $|n|$ input lists, and (b) the maximum length of a root-to-leaf path in T.*

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

4.4 Top-k query Processing

Our query model returns the k best matches to a user query. We adapt an existing and popular algorithm for efficiently finding the top k answers called the Threshold Algorithm (TA) [12]. *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, and dynamically accesses the sorted lists until the threshold condition is met to find the k best answers without evaluating all possible matches to a query. In our model, each list represents the answers to one of the path query decomposition of the twig query *TQ*. By traversing each path query DAG to access its matches in increasing degree of relaxations and using *TF* scores as tie breaker, we can produce a sorted list of matches for each path query. This is ensured by the monotonicity of *IDF* scores.

In the case where the query *TQ* decomposes in a single path query, a simple DAG traversal, with query pattern matching along the (more and more) relaxed versions of the query will yield the answers to *TQ* sorted by scores. If *TQ* decomposes to more than one path query, then each individual path query result is seen as a sorted list on which *TA* is applied. In addition to sorted access to the list, which is provided by a top-down DAG traversal, *TA* requires random access to individual scores. We adapt *RandomDAG* [22] to support random access. To this end, we must be able to determine if a query term appears in the pathname or content of a file. Specifically, we build a hash table for the terms in the original query to store the set of files that contain a given term. We then apply the *RandomDAG* algorithm to obtain the score of a file by dropping the query terms that do not appear in its pathname or content.

5 Experimental Evaluation

In this section, we explore the advantages offered by our unified structure and content search approach. Specifically, we first consider several example search scenarios, where a user is looking for a particular file within a relatively large personal data set in each scenario. We formulate a number of queries for each scenario and compare the ranks of the target file returned by our approach against those returned by a state-of-the-art desktop search tool. Next, we consider a much larger set of search scenarios using automatically generated queries. Finally, we report the query processing performance of and indexing space required by our unified search approach.

5.1 Experimental Setup

Relevance comparison. We use the Lucene text search engine [4] as a comparison basis. Specifically, we compare our approach against three different approaches: content-only and two variations of content and directory path terms. For *content-only*, we use the standard Lucene content indexing and search. For the first variation of content and directory path terms (*content:dir*), we create two Lucene indexes, one of content terms and one of terms from the directory pathnames; effectively, the latter treats each directory pathname as a file with the terms (components) in the pathname being its content. Then, each query can contain two conditions, one for content and one for directory path terms. Each query condition is scored individually against the appropriate index using Lucene. The scores are then combined using a vector projection approach as described in [19]. For the second variation (*content+dir*), we create a combined index that contains all content terms as well as directory path terms; terms in the pathname of each file is added to its content. Queries then contain terms that may match content or directory path terms. Queries are executed as searches against the combined index using Lucene.

We compare our unified approach to content:dir and content+dir because the latter two are plausible approaches that use some structure information (i.e., terms extracted from directory pathnames and internal structure) but are simpler to implement. Collectively, we refer to content-only, content:dir, and content+dir as "bag-of-terms" approaches because they do not consider structural relationships. We do not compare unified search against filtering approaches because the work in [19] has already shown that a flexible approach can find and rank relevant files that are missed entirely when filtering.

Data set. As noted in [11], there is a lack of synthetic

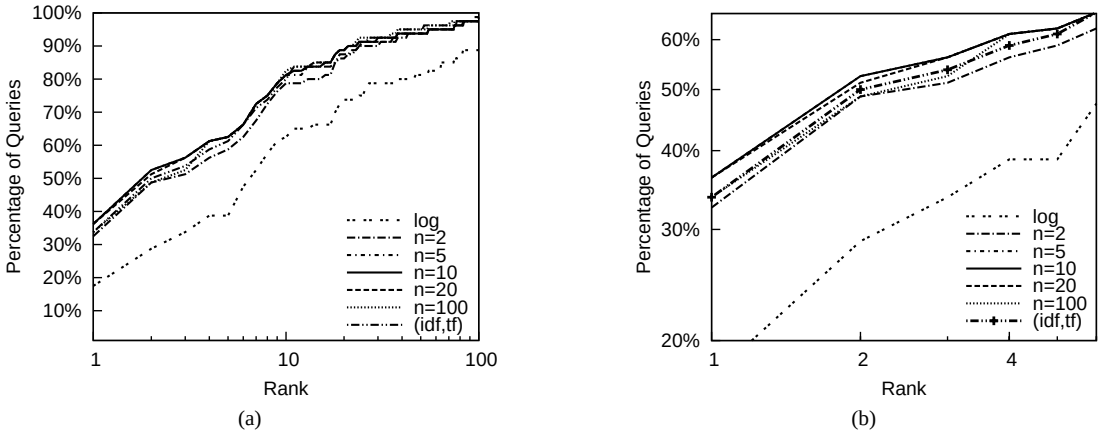


Figure 3: (a) CDFs of ranks of target files for six different *TF* formulas ($f(x) \in \{log(1+x)\} \cup \{x^{\frac{1}{n}} | n = 2, 5, 10, 20, 100\}$) and the approximation to unified ranking based on the lexicographical (*idf,tf*) ordering. (b) An enlarged region in plot a. The curves $n = 5$ and $n = 10$ almost overlap with each other, although $n = 10$ gives slightly better ranking.

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 contained 95,172 files in 7,788 directories; 6% of this data set were multimedia files (e.g., music and pictures), 59% document files (e.g., LaTeX, pdf, text, xml, and MS Office), 34% email messages[3], and 1% miscellaneous files (e.g., script files, source code, etc.). The average directory depth was 6.3 with the longest being 12. On average, directories contained 13 sub-directories and files, with the largest–a folder containing unsorted backup emails–containing 5,785. The system extracted close to 700K unique stemmed content terms and close to 3K unique directory path terms. The unified (directory and file) structure contained $\sim$57M nodes, of which $\sim$49M (86%) were leaf content nodes.

Platform. Experimental results for unified search are reported using a prototype tool implemented in Java. The tool uses the Berkeley DB [20] to persistently store all indexes. 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 1.5.0 JVM. Reported query processing times are averages of 40 runs, after 40 warm-up runs to avoid measuring JIT effects. All caches (except for any Berkeley DB internal caches) are flushed at the beginning of each run.

[3]Email messages were retrieved from the mail server through the IMAP interface. Each email was stored in a separate file in a directory hierarchy matching the folder hierarchy on the mail server.

5.2 Approximation to the Unified Ranking

We first experimentally identify the best unified scoring formula for our *IDF* and *TF* definitions. To this end, we used the *Unified* query set described in Section 5.1. We brute-forcefully computed *IDF* and *TF* scores for answers to all the queries contained in each relaxation DAG associated with the query set. The scores were combined across all (relaxed) queries to produce unified scores as detailed in Section 3.2. We ranked the documents based on their unified scores.

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

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

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

a rank of 3 for the target file. Figure 3 presents CDFs for seven different variations, six where the *TF* component of unified scores were computed using either logarithm or n-th root functions, and one where documents were sorted based on lexicographical (*idf,tf*) ordering with respect to the least relaxed query that a document matches.

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

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

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

Since our approximation to unified ranking is tight and it is practical to compute by only evaluating the most specific query that a file matches, we use this approximation approach throughout our implementation. For the rest of this chapter, we assume (without explicit mentioning) the approximation to unified ranking is always used for comparison between unified search and other search approaches.

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

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

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

Table 1 shows relative standard deviations of *IDF* and *TF* for six functions over the *Unified* query set. We only modify *TF* scoring to explore different impact of *TF* relative to *IDF*. Clearly *TF* has the smallest impact for logarithm function. As n increases for n-th root functions, *TF* attains bigger impact. When the whole document set

is considered, *IDF* has bigger impact for five functions. The relative standard deviations of *IDF* are same since *IDF* scoring formula is fixed. If top 100 documents are considered, the relative impact of *IDF* drops and it has bigger impact for four functions. There are small variations in relative standard deviations of *IDF* since the set of top 100 documents varies for different unified scoring formulas.

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

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

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

5.3 Case Studies

We now compare the relevance of unified search with "bag-of-terms" approaches. We start by comparing unified search with different search techniques based on several case studies.

Table 2 (at the end of this report) shows queries constructed for three different search scenarios and the resulting ranking by the four different search techniques. The target files are highlighted in Figure 1 to give an idea of how they are placed within the data set. The queries are meant to be representative of realistic queries composed by real users. A number of the queries contain inaccuracies representing when users may misremember information about what they are searching for.

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

A small amount of structure information can significantly improve search accuracy. In the absence of errors, U always achieve significantly better ranking for the target files than C. C:D also outperforms C for search scenarios 1 and 2. For example, U and C:D both achieve a rank of 1 for queries Q1 and Q6 compared to a rank of 20 achieved by C for query Q4. In search scenario 3, C (Q22) is better than C:D (Q24) because the structure terms *email* and *subject* do not add much differentiating power.

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

Structural relationships provide additional differentiating power for improving search accuracy. In search scenario 2, U was able to leverage the relationship in the *subject/"spring"* part of the query condition to achieve a ranking of 1-4 for the target file (Q18), whereas the inclusion of *email* and *subject* actually caused C:D to perform worse than C. *email* did not add much differentiating power while *subject* was only effective when considered together with the term "spring".

It is important to be able to relax query conditions across the content and structure dimensions. While it is important to differentiate between content and structure terms, it is also important to be able to relax across these two dimensions. This is because the user may not always remember correctly which are content and which are structure terms. When they are forced to explicitly identify content vs. directory terms, without relaxation support between the two dimensions such as in C or C:D, a mistake can drastically affect the search results. For example, switching a content and structure term drops the rank of the target file from 1 (Q6) to 245-252 (Q7) and from 4 (Q15) to 291 (Q16). On the other hand, U's processing of queries Q3 and Q12, which contain the same errors, rank the target file 1 and 1-3 respectively.

5.4 Automatically Generated Queries

To provide a more comprehensive evaluation of unified search, we also evaluated automatically generated queries

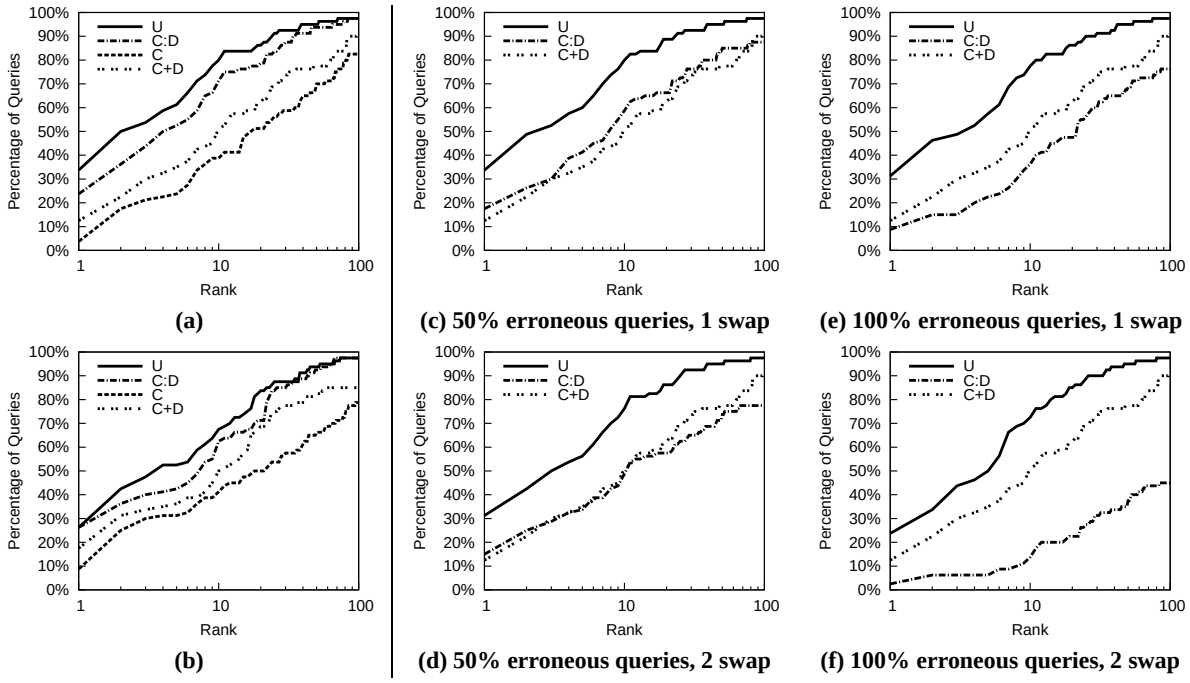


Figure 4: CDFs of ranks of target files for four different search techniques (a-b) and when queries contain inaccuracies (c-f). In (a) content terms were selected from those close to selected internal structure terms, while in (b) content and internal structure terms were selected independently. In (c-f), for U:C, U, and C:D, each figure contains a curve for 80 queries containing 50 or 100 percent erroneous queries. Each erroneous query switches one or two randomly chosen pairs of directory path and content terms. C+D is also shown as a baseline.

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

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

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

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

[4]We also constructed additional query sets where content was emphasized more heavily. The results for these query sets show similar trends.

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

Content+Dir: Each query contains all n terms.

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

Figure 4(a-b) plots the cumulative distribution function (CDF) of the rank of target files for all 80 queries, where each data point on the curve corresponds to the percentage of queries (Y-axis) with the corresponding target files positioned at or higher than a particular rank (X-axis). When there are ties, we use the median value of the range as the rank of the target file; e.g., 5 files, including the target file, achieve the same highest score would lead to a rank of 3 for the target file. Figure 4(a-b) presents CDFs for two different variations, one where content terms in the queries were selected to be close to selected internal structure terms (e.g., contained in a child content node of a selected internal structure node), and one where content and internal structure terms were selected independently.

These results reinforce the first two observations that we made in the previous section: (1) A small amount of structure information can significant improve search accuracy; and (2) It is important to distinguish between structure and content; Specifically, in Figure 4a, U and C:D always outperform C in ranking the target files; e.g., 80% of U queries and 71% of C:D queries ranked the target files 10 or higher, while only 39% of C queries ranked the target files within 10 or higher. Further, U and C:D queries are always better than C+D queries; e.g., only 51% of C+D queries ranked the target files 10 or higher.

Figure 4(a-b) also reinforces that *unified search leverages the relationships embedded in query conditions to outperform "bag-of-terms" approaches.* In particular, Figure 4(a-b) shows that U outperforms C:D when internal structure and content terms are chosen close together so that relationships embedded in the query conditions are meaningful. On the other hand, their performance gets closer when content terms and internal structure terms are chosen independently, effectively having less relationships between structure and content available for U to improve its ranking accuracy.

Finally, we consider what happens if queries contain inaccuracies, where external structure and content terms are mistakenly interchanged. Figure 4(c-f) plots CDFs of the rank of target files when inaccuracies are introduced into the queries. We consider inaccuracies along two dimensions, the percentage of queries (chosen randomly) containing inaccurate query conditions, and the level of inaccuracy, expressed as the number of switches between pairs of external structure terms and terms from inside the target file.

These results reinforce our final observation in Section 5: it is important to be able to relax conditions across the content and structure dimensions. Both U's and C:D's ranking performance degraded as the number of inaccurate queries and/or the number of inaccuracies in each inaccurate query increase. However, U queries are much less sensitive to the inaccuracies than C:D queries. In fact, Figures 4e and 4f show that C:D's can become significantly worse than C+D. On the other hand, even if 100% U queries have two pairs of directory path and content terms switched, U still outperforms C+D by a wide margin.

Thus, we conclude that our unified structure and content search approach has the potential to significantly improve search accuracy over existing "bag-of-terms" methods, even if the latter were extended to explicitly consider terms extracted from structure information. Specifically, the unified approach leverages term information from both structure and content as well as relationships between the terms to improve search accuracy. It is also robust against labeling inaccuracies (i.e., structure terms identified as content and vice versa) in query conditions.

5.5 Query Processing Performance

Figure 5 plots the CDF of the query processing times for the query set considered in Section 5.4. (We also measured processing times for the additional query sets mentioned in Section 5.4 and for queries considered in Section 5.3. These times are similar to those reported in Figure 5.) These results show that structure-heavy queries–recall that these queries on average contain 3.3 structure terms vs. 1.6 content terms–can increase query processing times. However, over 70% of the queries still require less

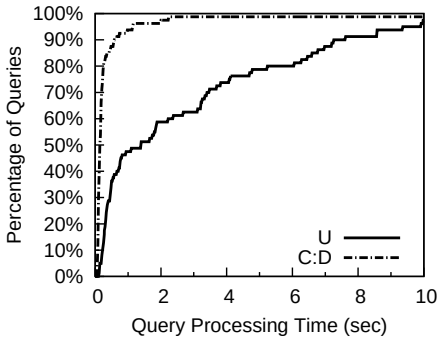


Figure 5: The CDF of query processing times to find and ranking top 10 relevant files for queries considered in Section 5.4.

than 4 seconds processing time, with the longest requiring 12 seconds.

There are several optimizations that can help to reduce query processing times. First, U is heavily penalized when queries include high frequency terms such as *title* and *subject* because they can significantly increase the time required to execute U's path query matching algorithm. This large penalty is mostly attributable the naive manner in which our index uses the Berkeley DB and can be significantly optimized with some coding effort. Second, there are opportunities for skipping unnecessary computation when processing U queries that we are not currently taking advantage of. We are in the process of incorporating these optimizations.

5.6 Storage Cost

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

6 Related Work

Several works have focused on the user perspective of personal information management [7, 17]. 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 [11, 25] 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. Our data model can be viewed as an XML data tree.

Integrating content and structure score is a complex task that requires a better understanding of the interconnections between structure and content. Efforts in the XML community have been made in this direction [14, 8] on a simpler set of structural relaxation rules; our techniques use new relaxation rules to achieve finer grained control to searching personal file system.

XML structural query relaxations have been discussed in [1, 3, 2]. Our work uses ideas introduced in the XML context, such as the DAG indexing structure to represent all possible structural relaxations [2], or the relaxed query containment condition [3, 2]. Our techniques differ from these work as we consider relaxations that convert content query nodes to structure query nodes and vice-versa, which allows searches across the file boundaries in a directory-based file system.

Our query evaluation approach relies on query matching. A previous approach *PathStack* [6] uses a chain of linked stacks to answer path queries. Partial path queries [23] extend this and allow fuzziness in the query conditions by keeping some structural relationships between query nodes undefined. While our work is also based on *PathStack* we support more complex path queries containing term permutations using dedicated query structures.

Learning and query selectiveness based ranking techniques for desktop search are proposed in [9]. Their ranking formula uses a linear functions to aggregate weights for various file features (filename, size, date of creation, etc.). The work in [19] computes separate scores for the content and structure dimensions and then aggregate

them into a single score. This work only considers external structure, whereas we unify external and internal structure. Further, similar to C:D (Section 5), cross-dimensional mistakes in queries can deteriorate search results. In contrast, as already shown, our unified approach is robust to such mistakes.

The user study [5] has shown that location based search strategy largely dominates the recall-directed search phase and the most frequently free-recalled attributes are the characteristics of the documents' textual content (e.g., abstract, structure, distinctive portions of text like the title etc.). Therefore, both the external directory structure and internal file structure play important roles for the search of personal information. Furthermore, these studies conclude that systems should take into account the approximation of the recall in the returned results by allowing errors in the query parameters. Our work addresses this issue by relaxing the query conditions automatically during search.

7 Conclusion and Future Work

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

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

References

[1] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree Pattern Relaxation. In *Proc. of the International 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 International 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 International Conference on Management of Data (SIGMOD)*, 2004.

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

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

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

[7] 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.

[8] 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.

[9] S. Cohen, C. Domshlak, and N. Zwerdling. On Ranking Techniques for Desktop Search. *ACM Transactions on Information Systems (TOIS)*, 26(2), 2008.

[10] W. B. Croft, P. Krovetz, and H. Turtle. Interactive retrieval of complex documents. *Information Processing and Management*, 26(5), 1990.

[11] 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.

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

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

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

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

[16] T. Grust. Accelerating XPath Location Steps. In *Proc. of the ACM International Conference on Management of Data (SIGMOD)*, 2002.

[17] 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.

[18] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

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

[20] Sleepycat Software. Berkeley DB. http://www.sleepycat.com/.

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

[22] W. Wang, C. Peery, A. Marian, and T. D. Nguyen. Efficient Multi-Dimensional Query Processing in Personal Information Management Systems. Technical Report DCS-TR-627, Department of Computer Science, Rutgers University, 2008.

[23] X. Wu, S. Souldatos, D. Theodoratos, T. Dalamagas, and T. Sellis. Efficient Evaluation of Generalized Path Pattern Queries on XML Data. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, 2008.

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

[25] 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.

Query Number	Query Type	Query Conditions	Comment	Rank
Search Scenario 1: The user searches for a picture of Alice wearing a witch costume taken at home on Halloween. **Target file:** /Desktop/Pictures/Disk 3/2008/Home/20081101/IMG_1391.gif (tagged with "witch" and "halloween")				
Q1	U	//home[.//"witch" and .//"halloween"]	Accurate query conditions	1
Q2	U	//home/alice[.//"witch" and .//"halloween"]	Extraneous structure condition	1
Q3	U	//halloween/witch/"home"	Structure and content terms switched	1
Q4	C	{witch, halloween}	Accurate query conditions	20
Q5	C	{home, witch, halloween}	Structure term used as content	31
Q6	C:D	{witch, halloween} : {home}	Accurate query conditions	1
Q7	C:D	{witch, home} : {halloween}	Structure and content terms switched	245-252
Q8	C+D	{home, witch, halloween}	Accurate query conditions	20
Search Scenario 2: The user searches for the chapter "Of the Travelling of the Utopians" in the electronic book "Utopia". **Target file:** /Laptop/eBooks/Non-Fiction/Philosophy/Utopia/OPS/main6.xml				
Q9	U	//philosophy[.//"utopia" and .//"travel"]	Accurate query conditions	1-2
Q10	U	//philosophy[.//"utopia" and .//chapter/"travel"]	Extraneous structure condition	1-2
Q11	U	//title[.//philosophy/"utopia" and .//"travel"]	Out-of-order structure conditions	1-2
Q12	U	//utopia/travel/"philosophy"	Structure and content terms switched	1-3
Q13	C	{utopia, travel}	Accurate query conditions	18
Q14	C	{philosophy, utopia, travel}	Structure term used as content	9
Q15	C:D	{utopia, travel} : {philosophy}	Accurate query conditions	4
Q16	C:D	{philosophy, utopia} : {travel}	Structure and content terms switched	291
Q17	C+D	{philosophy, utopia, travel}	Accurate query conditions	24
Search Scenario 3: The user searches for an email with the subject "Spring 2006 Tuition Payment ...". **Target file:** /Laptop/email/local/Backup/2005_Mail_Backup/Inbox/324.xml				
Q18	U	//email[.//subject/"spring" and .//"bill"]	Accurate query conditions	1-4
Q19	U	//email/department[.//subject/"spring" and .//"bill"]	Extraneous structure condition	1-4
Q20	U	//email[.//subject/"bill" and .//"spring"]	Out-of-order structure conditions	3-35
Q21	U	//email[.//spring/"subject" and .//"bill"]	Structure and content terms switched	1-4
Q22	C	{spring, bill}	Accurate query conditions	36
Q23	C	{email, spring, bill}	Structure term used as content	340
Q24	C:D	{spring, bill} : {email}	Accurate query conditions	70
Q25	C:D	{subject, spring, bill} : {email}	Accurate query conditions	142
Q26	C:D	{bill} : {email, spring}	Structure and content terms switched	596-635
Q27	C+D	{spring, bill, email}	Accurate query conditions	75
Q28	C+D	{subject, spring, bill, email}	Accurate query conditions	153

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