Unified Structure and Content Search for Personal Information Management Systems*

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

ABSTRACT

User data stored in personal information systems is growing massively. Simultaneously, this data is increasingly distributed across multiple organizational domains such as email, music databases, and photo albums, some of which are structured automatically by applications. Powerful search tools are needed to help users locate data in these 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 framework uses unified data and query processing models so that structure conditions can be approximately matched by content and vice versa. Our models also unify external structure (e.g., directories) with internal structure (e.g., XML structure), supporting integrated queries matched to a single data domain. We propose indexes and algorithms for efficient query processing. We evaluate our approach using a real data set, showing that it can leverage structure information to significantly improve search accuracy, yet is robust to mistakes in query conditions.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information filtering, Search process*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Personal information search, structure and content search, query processing, query path matching

1. INTRODUCTION

The amount of data stored in Personal Information Management systems (PIMs) is rapidly increasing, following the relentless growth in capacity and dropping price of storage. This data explosion is driving a critical need for search tools to retrieve 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 for file systems, including the commercial tools Google Desktop Search [15] and Spotlight [20]. However, most of these tools 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 to Personal Information and Dataspaces [7, 11, 13], which consist of heterogeneous data collections. 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 do not exploit the rich structural information typically available in PIMs. Unlike searches over digital libraries and the Web, users searching their personal files frequently have some recall of file locations (directory structure) and the structure of content inside files such as title and abstract (internal structure) [5, 10]. However, this recall is typically imperfect because structure information is large and complex, evolves over time, and may be programmatically organized (e.g., photo management software).

Thus, it is too rigid to use this 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. *A user John wants to retrieve photos of a Halloween party held at his home where someone was wearing a witch costume from his personal file system.*

Ideally, the directory structure would have been created and maintained consistently and all photos properly tagged. In practice, this is rarely the case: users change their file organizations over time, inconsistently annotate their data, use applications that (re)arrange their data on disk, and gather data from different sources. In our example, John has changed the way he organizes his photos over time and do not always tag his pictures. As a result, pictures from different Halloween parties match very different directory structures and do not necessarily have matching tags, as illustrated in Figure 1. In addition, some relevant pictures are in his email folder because they were sent to him by friends.

This structural heterogeneity complicates the search for specific pictures. A content-only search for "Halloween, home, witch" is likely to result in many matches. None of the pic-

*This research was partially supported by NSF grants IIS-0844935 and CNS-0448070.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT 2011, March 22–24, 2011, Uppsala, Sweden.
Copyright 2011 ACM 978-1-4503-0528-0/11/0003 ...$10.00

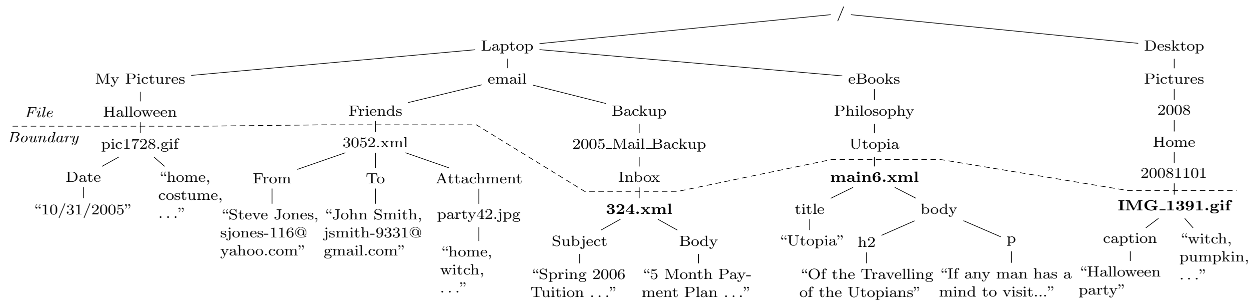


Figure 1: A partial example user personal information file system.

tures, pic1728.gif, party42.jpg, and IMG_1391.gif, *in the example data set 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.*

Fortunately, John vaguely remembers that the photo he is looking for is in his home directory and was annotated with a caption containing the term "Halloween". Based on this information, John can write the 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:*

- *Using the structure part* `//home//caption` *as a filtering condition would eliminate files that do not match it exactly. For example,* party42.jpg, *which contains the keywords "witch" and "home" but does not have a caption tag with value "Halloween" would not be returned.*
- *Since the external (directory structure) and the internal (structure and content) are strictly separated, answers that do not adhere to this strict separation would be missed. For example,* pic1728.gif *would not be returned since "Halloween" is expected to be a content term and not part of the structure hierarchy.*

Because of the structural and data heterogeneity present in PIMs, 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 external structure, internal structure, and content into 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 approximately 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.

We make the following contributions:

- We propose unified data and query models that allow fuzzy matching of each query condition 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 that extends the popular *PathStack* algorithm [6] to handle 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).

2. DATA AND QUERY MODEL

2.1 Unified Data Model

We model the entire file system as a rooted, labeled, unordered tree that contains internal *structure* nodes and leaf *content* nodes. Each structure node has a label that is used to record the name of the structure item (e.g., directory name) or its type within some structural schematic (e.g., section in a LaTeX document). Each content node contains a label that is used to record the content term the node represents. In the rest of the paper we refer to this data representation as the *unified data tree* T.

Figure 1 shows a partial unified data tree for an example user personal file system. Each node is shown by its label. The external structure (directories) and internal structure (e.g., the "from" field in an email or "title" of an ebook) of files are both represented as internal structure nodes in the unified data tree.[1] Content is stored in the leaves. Abstractly, each leaf node only contains one term although in the implementation, sibling content nodes are combined together to save space. (The dotted line representing the file boundary is given in the figure for illustration purpose only.)

To simplify the discussion, we leave out file system metadata information such as file size and modification time in this paper. Metadata can easily be included in the unified data tree, with both internal structure nodes (e.g., "Last Modified") and leaf value nodes (e.g., "10/31/09"). External tags (e.g., Spotlight comments) can be similarly handled. Tags used for organization (e.g, Gmail "folders") can be treated just like directories although we would currently have to (re)index a file for each tag to maintain a tree structure.

[1]We could have separated the two types of structures. However, we don't think it is profitable to do so because it would require users to understand the structural representation of data, which is unrealistic since this representation is increasingly being managed by applications.

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 patterns and content terms that can be represented as a twig, in the spirit of XQuery [24]. However, as previously mentioned, structural query conditions are likely incomplete and may contain mistakes. Thus, our query model allows for approximate data matches for both structure and content conditions (Section 3) to avoid discarding relevant information because of (possibly minor) mistakes in the query.

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

A **root node**, noted *root*, is a node in the twig query (query node) that is matched by the root of the unified data tree T. *root* may only appear at the beginning of a query.

A **structure node**, noted *N*, is a query node that can be matched by any internal structure node with label N in T.

A **content node**, noted *"N"*, is a query node that can be matched by any leaf node with label N in T.

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

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

A **path segment** *PS* is a partial path where each node is either a structure, content, generalized, 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 unique node in P and the matching nodes in P preserves the edge structure of *PS*.

A **node group** (similar to that introduced in [19]) is used to represent possible permutations of query nodes. Specifically, a node group, noted *(PS)*, is a path segment *PS*, 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 segment although the node labels may permute. Root nodes, content nodes, and extended nodes are not allowed in a node group as they can only occur at the beginning or the end of a query path. A node group *(PS)* can be matched by any path in T that matches a valid permutation of *PS*.

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

Extending a node group is the same as extending each path segment in its permutation set.

Our model considers twig queries over the unified data tree T. A twig query is a tree that starts with a root node *root* and may contain multiple branches. Branches may end with any node type defined above. Generalized, content, and extended nodes can only be positioned as the last node of a branch.

Note that keyword-only queries can be easily specified in our model. For example, a query with keywords $k1$ and $k2$ would be specified as //[.//"$k1$" and .//"$k2$"]. The returned results, however, may be (slightly) different than in a typical content-only search. As shall be seen, the keywords may match external structure terms because of query relaxations and our scoring functions are not strictly the same as those used in traditional $TF{\cdot}IDF$ content-only search approaches. Experimentally, however, we observe very little differences between our system and a typical content-only search tool for keyword-only queries.

In this paper, we focus on 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 path query *PQ* can be matched by any path *PM* in a data tree *T* where each node in *PQ* is matched by a unique node in *PM*, and the matching nodes in *PM* preserves the edge structure of *PQ*.

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 (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 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 dimensions, as well as across the two dimensions. We score individual paths of a given query twig in a $TF{\cdot}IDF$-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 transformations that make queries more general. 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 [3, 19] on structural query relax-

ations 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 structure and content 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. For example, applying edge generalization to the path segment *home/Halloween* changes it to *home//Halloween*.

Path Extension is used to extend a path query PQ to *PQ//** so that any path in T containing PQ becomes a match. For example, applying path extension to */home//Halloween* changes it to */home//Halloween//**.

Node Generalization is used to relax a structure node at the end of the path query or a content node to a generalized node. This relaxation allows structure conditions to be approximately matched by content and vice versa. For example, applying node generalization on */home/Halloween* changes it to */home/{Halloween}*, which means the term "Halloween" in the query can match the name of a directory, the tag of an internal structure item, or file content.

This novel relaxation can be critically important in our search context since content terms may often be used as structure (e.g., directory names), especially when data is programmatically organized. For example, iTunes organizes users' music in directories named after artists and albums.

Node Inversion is used to permute nodes within a path query PQ. Except for the root, non-generalized leaf, and * nodes, inversion can be applied to any pair of adjacent nodes or node groups if all the surrounding edges of the nodes or node groups are ancestor-descendant edges. An inversion combines adjacent nodes, or node groups, into a single node group while preserving the placement of the generalized node, if any. For example, applying node inversion to *Halloween* and *witch* in */home//Halloween//witch* changes it to */home//(Halloween//witch)*. Applying the same node inversion on */home//Halloween//{witch}* changes it to */home//(Halloween//{witch})*, which allows for */home//witch//{Halloween}* in addition to the original query condition.

Allowing for approximations in the node ordering can be critically important in our search context since, as already mentioned, users often misremember, or do not have complete information over, the directory or internal file structure, yet users often know some pertinent structural information that can help guide the search.

Node Deletion is used to drop a node from a path query. Node deletion can be applied to any node or node group other than *root* and * as long as their surrounding edges are ancestor-descendant edges.

To delete a node n in a path query PQ:

- If n is a leaf node, n is dropped from PQ and PQ–n is extended with *//**. This is to ensure containment of the answers to PQ in the set of answers to the new, more relaxed path query *PQ'*.
- If n is an internal node, n is dropped from PQ, and the nodes before and after n are connected in PQ with //.

For example, deleting *Halloween* from */home/Halloween/witch* changes it to */home//witch*.

To delete a node n inside a node group *(PS)* in a path query PQ, the following two steps are required to ensure answer containment. (1) n and one of its adjacent edge in PS are dropped from PS. If only one node m is left in PS, *(PS)* is replaced by m in PQ. (2) If *(PS)* is a leaf node group, the resulting query is extended so that permutations of the original node group *(PS)* that contain n as a leaf node are still contained in the resulting relaxed node group.

For example, deleting *Halloween* from *//(home//Halloween//{witch})* changes it to *//(home//{witch})//**. The path query *//(home//Halloween//{witch})* contains 6 different permutations, including *//home//Halloween//{witch}*, *//witch//Halloween//{home}*, and *//home//witch//{Halloween}*; after deleting *Halloween*, the permutations become *//home//{witch}*, *//witch//{home}*, and *//home//witch//**. *//(home//{witch})//** is the only most specific path query that contains all of the necessary permutations to ensure containment.

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.

Note that while the above relaxations could be used to tolerate mistakes like misspelled terms, they are not designed for that. In a production search tool, they would be combined with other complementary IR techniques. For example, one can envision each term in a query condition matched against a set of terms to account for misspelling, stemming, and/or even semantic equivalency.

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 before aggregating them into a single score, our approach scores content and structure 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 files matching more relaxed forms of a query will receive lower scores since more relaxed forms always match the same number or more files based on the containment property of relaxations.

Definition 2 (TF Score of a File for a 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 numbers of structure and content nodes in F, respectively, $F_{PQ}^{(struct)}$ and $F_{PQ}^{(cont)}$ are the numbers of structure and content match points in F, respectively, 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.*

By definition, the more match points contained in an answer, the higher the *TF* score of the answer. (We can potentially aggregate the two scores in different ways. In fact, we tried aggregating using multiplication and found experimentally that it made little difference.)

In Definition 2, function f is selected from a class of functions (e.g., those widely used for *TF* in the IR community). In a real system, function f is selected once and then fixed for all queries and answers to compute *TF* scores. Different choices of f change the distribution of score values, affecting the relative impact of *TF* vs. *IDF* on the unified scores. We experiment with different variants of f to find the best scoring formula for our data set in Section 5.2. (We could potentially use different functions for content and structure; currently, we use the same function for simplicity.)

Definition 3 (Score of a File for a 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. Abstractly, 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. 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 very expensive to compute $score(PQ, F)$ as defined in Definition 3 because the number of relaxed forms of a query grows exponentially with query size. Thus, our implementation instead approximates the definition using a lexicographical ordering given by $(score_{idf}(LRQ_F^{PQ}), score_{tf}(LRQ_F^{PQ}, F))$, where LRQ_F^{PQ} is the least relaxed query, i.e., the relaxed query with the highest *IDF* score, that a file F matches. We empirically show the tightness of this approximation for the data and query sets used in our evaluation (Section 5.2). The algorithms and data structures described in the following section are based on this approximate scoring to speed up query processing.

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 to the nodes' attribute tuples. Note that this index subsumes the typical full-text inverted index used for content search because each content term is a label of a leaf node (Section 2).

Indexing Query Relaxations: We represent (abstractly) 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). The most relaxed version of any query condition is the node *//**, which matches all files and so gives a 0 *IDF* score.

The above DAG is built *lazily* during query processing; that is, we only build and evaluate parts of the DAG as required by the top-k query processing algorithm (Section 4.3). *IDF* scores for relaxed queries and *TF* scores for matching files are only computed when nodes are materialized. Previous work have detailed efficient algorithms for this lazy construction [2, 19, 22]. We adapt these algorithms to handle the full set of relaxations described in Section 3.1.

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.

The inverted data index is used to avoid traversal of the data tree. Specifically, each query node is associated with the inverted list keyed by the node's label. Preorder traversal is then simulated by considering the nodes from the

Algorithm 1 *NIPathStack(NG)*

```
1.  Tail ⇐ nil
2.  while ¬end(NG) do
3.    N_min ⇐ getMinSource(NG)
4.    while Tail ≠ nil ∧ postCode(Tail) < nextPre(T_{N_min}) do
5.      pop(S_Tail)    {S_Tail is the stack containing node Tail.}
6.      Tail ⇐ prevDataPathNode(Tail)
7.    moveNodeToStack(T_{N_min}, S_{N_min}, pointer to Tail))
8.    Tail ⇐ top(S_{N_min})
9.    if containSolution(NG) then
10.     NIShowSolutions(Tail, NG)

11. function end(NG)
12. begin
13.   return ∀N_i ∈ NG : eof(T_{N_i})

14. function containSolution(NG)
15. begin
16.   return ∀N_i ∈ NG : |S_{N_i}| ≥ 1

17. function getMinSource(NG)
18. begin
19.   return N_i ∈ 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)
```

matching lists in sorted order according to their *PreCode*.

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

First, for each node group *NG* in a query *PQ* we use a new *NIPathStack* algorithm to find all matching path segments in the unified data tree. These matches are returned in increasing lexigraphical ordering given by the pair ($PreCode_t$, $PreCode_h$), where $PreCode_t$ and $PreCode_h$ are the *PreCodes* of the deepest (tail) and highest (head) nodes, respectively, of the matching path segment.

Second, we apply *PathStack* on *PQ* with a small variation: each node group *NG* is given an individual stack, and its matches are populated by the 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. The inverted data index is again used to simulate the preorder traversal without actually traversing the data tree. As nodes are pushed onto the stacks, *NIPathStack* maintain pointers between these nodes to represent 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*. Lines 4-6 pop stack nodes when the al-

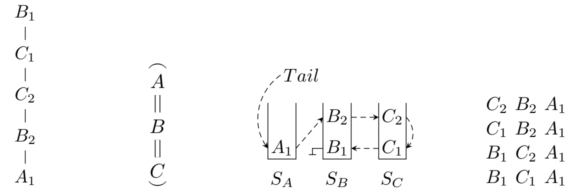
Algorithm 2 *NIShowSolutions(Tail, NG)*

```
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)
```

gorithm moves to the next branch of the unified data tree. The function *prevDataPathNode* is used to return the previous node in the data path encoded in stacks. Lines 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.

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 (lines 13 and 16) sequentially.



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

Figure 2: Example data path and answer encoding in stacks.

Figure 2c shows the stack encoding of the data path segment $B_1/C_1/C_2/B_2/A_1$ (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 deepest node (A_1), with all answers ending with this node produced recursively ($B_1/C_1/A_1$, $B_1/C_2/A_1$. $C_1/B_2/A_1$, 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.

4.3 Top-k query Processing

As previously mentioned, we decompose a twig query into its component path queries. Given an answer, we compute a score for the answer for each component path query. We then combine these individual scores into an overall score representing the answer's relevance to the twig query. This computation can be viewed as a multidimensional scoring

problem, with each component path query representing a distinct scoring dimension.

We adapt the existing and popular Threshold Algorithm (TA) [12] to efficiently solve our multidimensional scoring problem. 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 considering all objects.

In our adaptation, each sorted list contains answers for one component path query. We generate each list by traversing the relaxation DAG of the corresponding path query and finding matches to increasingly relaxed versions of the path query. The monotonicity of scores given by our scoring methodology (lexicographical ordering based on (idf, tf)) guarantees that this traversal will produce answers sorted in decreasing order of their scores. As already mentioned, we adapt (slightly) existing algorithms for lazily building and traversing the DAG, including an optimization to avoid scoring unnecessary nodes (i.e., nodes that do not contribute new answers) [22].

As TA considers the next answer from a sorted list, it needs to compute the overall score for the answer; i.e., the combine score across all attributes. This means that we also need to compute the score for an arbitrary answer to a path query. One possible approach is to expand the DAG until we locate the LRQ that matches the answer to be scored. This can be expensive, however, as it may require expanding deep into the DAG. Thus, we adapt an optimization from [22] that allows us to leverage the inverted data index to quickly jump to the LRQ or a close by ancestor.

5. EXPERIMENTAL EVALUATION

We now experimentally study and evaluate our unified search approach. Specifically, we first consider several example search scenarios, where a user is looking for a particular file within a 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 Lucene [4], a state-of-the-art desktop search tool. We choose Lucene because it is open-source, allowing us to adapt it to a range of approaches for using content and structure terms. Next, we consider a much larger set of search scenarios using automatically generated queries. We also use (part of) this larger query set to compare our approach with Google Desktop Search (GDS) and TopX [21], a related approach that was designed for XML search. 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 these Lucene-based approaches as "bag-of-terms" 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. We use a subset of files and directories from the working environment of one of the authors because there is a lack of synthetic data sets and benchmarks to evaluate PIMs search (as noted in [11]). This data set contains 95,172 files in 7,788 directories; 6% of the files are media (e.g., music and pictures)[2], 59% documents (e.g., LaTeX, pdf, and MS Office), 34% emails[3], and 1% miscellaneous (e.g., scripts and source code). The average directory depth is 6.3 with the longest being 12. Indexing leads to ∼700K unique stemmed content terms and ∼3K unique directory path terms. The unified data tree contains ∼57M nodes, of which ∼49M (86%) are leaf content nodes.

Query set. We manually construct 28 queries for 3 search scenarios for our first case study of unified search.

We also automatically generate queries for a larger set (80) of search scenarios to provide a more comprehensive evaluation. These scenarios comprise four sets of 20 scenarios each, targeting files from four different data categories, including email, document (ebooks, academic papers, etc.), music, and picture. Queries are constructed to contain varying numbers of query conditions, as well as different combinations of structure and content terms.

More specifically, each query targets a specific file f. Each query comprises n terms, where n is randomly chosen from $\{4, 5, 6\}$. The n terms are 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 is designed to select approximately $n/2$ external structure terms, $n/4$ internal structure terms, and $n/4$ content terms. Because some target files do not contain any internal structure, this process leads 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]

[2]ID3 and IPTC tag names (structure) and values (content) are extracted from music and pictures, respectively.

[3]Each email message is stored in a separate file in directories that match the user's mail folder hierarchy.

[4]We also studied additional query sets that emphasize content more heavily. The results for these query sets show similar trends.

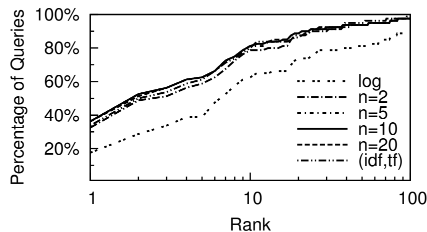


Figure 3: CDFs of ranks of target files when using Definition 3 with five different *TF* formulas ($f(x) \in \{log(1+x)\} \cup \{x^{\frac{1}{n}} | n = 2, 5, 10, 20\}$) and when using the lexicographical approximation (*idf, tf*).

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"*.

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.

Platform. We have implemented a prototype unified search tool in Java, using the Oracle Berkeley DB to persistently store all indexes. Experiments are run on a PC with a 2.8 GHz Intel Xeon processor, 2 GB of memory, and a 10K RPM 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.

5.2 Approximate Unified Ranking

To compute unified scores for our experiments, we must first choose a specific function f for *TF* (Definition 2). Figure 3 plots the cumulative distribution functions (CDFs) of the ranks of target files for the automatically generated query set described in Section 5.1, where each data point on a curve corresponds to the percentage of queries (Y-axis) with the corresponding target files positioned at or higher than a particular rank (X-axis). We study five different formulas, $f(x) \in \{log(1+x)\} \cup \{x^{\frac{1}{n}} | n = 2, 5, 10, 20\}$, which are common alternatives for *TF* from the IR community (e.g., [4, 18]), when using the full scoring formula defined in Definition 3. (We compute the scores by brute-force full expansion of the query DAGs.) We also study the approximate scoring function based on the lexicographical ordering (*idf, tf*) computed against the least relaxed query that a file matches, as described in Section 3.2.

We observe that the log function gives the worst result. When using the n-th root functions, the results are relatively insensitive to n (variations of less than 5% for n between 2 and 20) although n=10 does give the highest accuracy. All results reported in the rest of this section are obtained using the function $f(x) = x^{\frac{1}{10}}$ to compute TF scores.

To study the impact of *TF* vs. *IDF*, we show the relative standard deviation (100% x standard deviation / mean) for the two score components for different *TF* formulas in Table 1. We study the relative standard deviation because the scoring formula in Definition 3 multiplies *TF* and *IDF*. The larger the relative standard deviation, the greater the impact of the measure in differentiating between answers. Table 1 shows that the logarithm formula seems to overweight TF, especially when considering only the top 100 matches.

Table 1: Relative standard deviations of *IDF* and *TF* scores for the five different *TF* functions.

	Relative Standard Deviation (%)			
	All Files		Top 100 Answers	
Function	*IDF*	*TF*	*IDF*	*TF*
log	60.95	68.06	37.75	53.56
$n = 2$	60.95	45.33	36.63	40.76
$n = 5$	60.95	27.27	35.87	28.14
$n = 10$	60.95	20.20	35.75	23.80
$n = 20$	60.95	16.90	35.71	21.80

For comparison, we also computed the relative standard deviations of *TF* and *IDF* scores for Lucene. Interestingly, the results show that the impact of *TF* is greater than *IDF* when the whole document set is considered. However, the impact of *TF* is smaller than *IDF* when considering the top 100 documents, which is similar to the n-th root functions for our scoring definition with $n \geq 5$.

Figure 3 also shows that the lexicographical (*idf, tf*) ordering based on the least relaxed query that an answer matches is a tight approximation to the full scoring formula. For any ranking position, the difference between the CDF curves for (*idf, tf*) and the full scoring function with $f(x) = x^{\frac{1}{10}}$ is less than 3%. The rest of this section uses the lexographical approximation (*idf, tf*) for computing the unified scores.

5.3 Case Study

Table 2 shows queries constructed for three different search scenarios and the rankings of the target files returned 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. (Note that the actual pathnames given in Table 2 are somewhat longer because Figure 1 has been condensed to save space.) The queries are meant to be representative of realistic queries composed by real users. A number of the queries contain inaccuracies representing when users mistakenly identify structure terms as content and vice versa.

We make the following observations.

A small amount of structure information can significantly improve search accuracy. In the absence of errors, U always ranks the target files higher than C. C:D also outperforms C for scenarios 1 and 2. For example, U and C:D rank the target file for Q1 and Q6 1st compared to C's 20th ranking for Q4. In 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

Q. #	Q. Type	Query Conditions	Comment	Rank
Search Scenario 1: The user searches for a picture of Alice wearing a witch costume taken at home on Halloween. **Target file:** /Desktop/Pictures/Disk 3/2008/Home/20081101/IMG_1391.gif (tagged with "witch" and "halloween")				
Q1	U	//home[.//"witch" and .//"halloween"]	Accurate query conditions	1
Q2	U	//home/alice[.//"witch" and .//"halloween"]	Extraneous structure condition	1
Q3	U	//halloween/witch/"home"	Structure and content terms switched	1
Q4	C	{witch, halloween}	Accurate query conditions	20
Q5	C	{home, witch, halloween}	Structure term used as content	31
Q6	C:D	{witch, halloween} : {home}	Accurate query conditions	1
Q7	C:D	{witch, home} : {halloween}	Structure and content terms switched	245-252
Q8	C+D	{home, witch, halloween}	Accurate query conditions	20
Search Scenario 2: The user searches for the chapter "Of the Travelling of the Utopians" in the electronic book "Utopia". **Target file:** /Laptop/eBooks/Non-Fiction/Philosophy/Utopia/OPS/main6.xml				
Q9	U	//philosophy[.//"utopia" and .//"travel"]	Accurate query conditions	1
Q10	U	//philosophy[.//"utopia" and .//chapter/"travel"]	Extraneous structure condition	1
Q11	U	//title[.//philosophy/"utopia" and .//"travel"]	Out-of-order structure conditions	1
Q12	U	//utopia/travel/"philosophy"	Structure and content terms switched	2
Q13	C	{utopia, travel}	Accurate query conditions	18
Q14	C	{philosophy, utopia, travel}	Structure term used as content	9
Q15	C:D	{utopia, travel} : {philosophy}	Accurate query conditions	4
Q16	C:D	{philosophy, utopia} : {travel}	Structure and content terms switched	291
Q17	C+D	{philosophy, utopia, travel}	Accurate query conditions	24
Search Scenario 3: The user searches for an email with the subject "Spring 2006 Tuition Payment ...". **Target file:** /Laptop/email/local/Backup/2005_Mail_Backup/Inbox/324.xml				
Q18	U	//email[.//subject/"spring" and .//"bill"]	Accurate query conditions	3
Q19	U	//email/department[.//subject/"spring" and .//"bill"]	Extraneous structure condition	3
Q20	U	//email[.//subject/"bill" and .//"spring"]	Out-of-order structure conditions	19
Q21	U	//email[.//spring/"subject" and .//"bill"]	Structure and content terms switched	3
Q22	C	{spring, bill}	Accurate query conditions	36
Q23	C	{email, spring, bill}	Structure term used as content	340
Q24	C:D	{spring, bill} : {email}	Accurate query conditions	70
Q25	C:D	{subject, spring, bill} : {email}	Accurate query conditions	142
Q26	C:D	{bill} : {email, spring}	Structure and content terms switched	596-635
Q27	C+D	{spring, bill, email}	Accurate query conditions	75
Q28	C+D	{subject, spring, bill, email}	Accurate query conditions	153

Table 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. 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 so that the terms "travel", "travelling", and "traveling" are equivalent for search scenario 2.

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 scenario 2, U uses the relationship in the *subject/"spring"* part of the query condition to achieve a ranking of 3 for the target file (Q18), whereas the inclusion of *email* and *subject* actually causes 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, it is also important to be able to relax across the two dimensions. This is because users may not always remember correctly which are content and which are structure terms. When they are forced to explicitly identify content vs. directory terms, a mistake can drastically affect the search results if cross-dimension relaxation is not supported. For example, switching a content and structure term drops the rank of the target file from 1 (Q6) to 245-252 (Q7) and from 4 (Q15) to 291 (Q16). On the other hand, U's processing of queries Q3 and Q12, which contain the same errors, rank the target file 1 and 2, respectively.

5.4 Automatically Generated Queries

Figures 4a-b plot the CDFs of the ranks of target files for all 80 automatically generated queries (Section 5.1). 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. Figures 4a-b present CDFs for two different variations, one where content terms in the queries are 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 are selected independently.

These results reinforce the first two observations 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; e.g., 80% of U queries and 71% of C:D queries rank the target files 10 or higher, while only 39% of C queries rank the target files within 10 or higher. Computing the corresponding Mean Reciprocal Rank Values (MRR) at 10 gives 0.484 for U, 0.161 for C, 0.379 for C:D, and 0.239 for C+D. (U outperforms all techniques with a statistical significance at $p < 0.05$ using the Wilcoxon one-tailed test). Further, U and

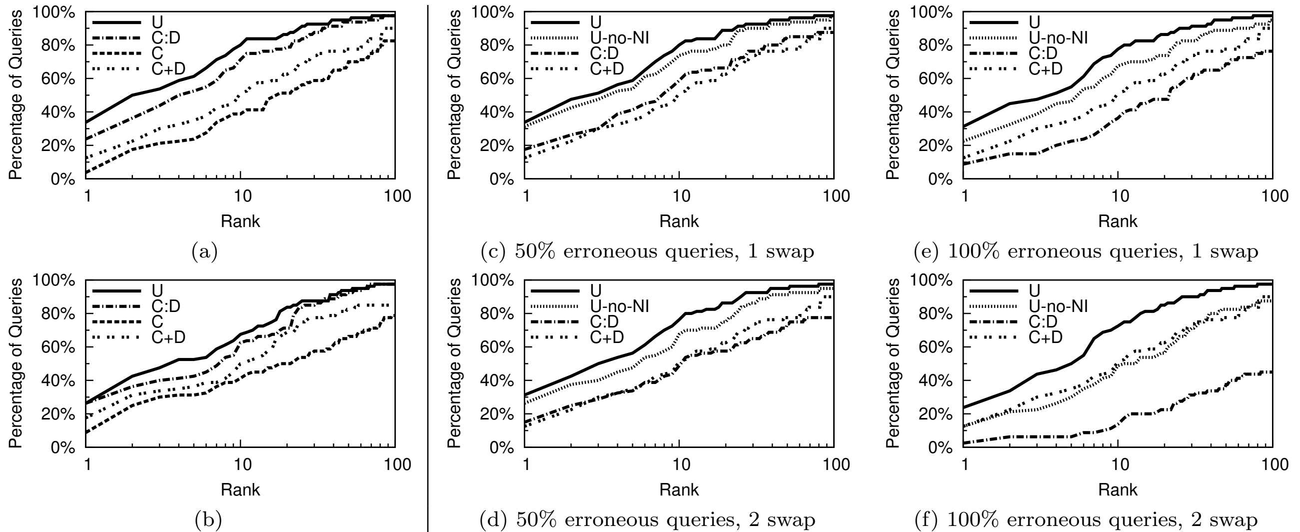


Figure 4: CDFs of ranks of target files (a-b), when queries contain inaccuracies (c-f). In (a) content and internal structure terms are selected "close" together, while in (b) they are selected independently. In (c-f), for U, U-no-NI, and C:D, 50 or 100 percent of the queries contain errors. Each erroneous query switches one or two randomly chosen pairs of directory path and content terms. C+D is shown as a baseline. U-no-NI refers to the use of U without the node inversion relaxation.

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.

Figures 4a-b also reinforce the third observation above: structural relationships provide additional differentiating power for improving search accuracy. In particular, the figures show 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. We choose this type of errors because it may be easy for users to confuse structure and content terms, particularly when parts of the namespaces are programmatically organized. 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. Figures 4c-f plot CDFs of the ranks of target files when inaccuracies are introduced into the queries. These graphs include results when U is used without the node inversion relaxation (U-no-NI) to evaluate the importance of node inversion to our search approach.

The results shown in Figures 4c-f reinforce our final observation in Section 5.3: 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. This is reflected in the corresponding MRR: when the percentage of query errors increases to 50% of all queries (2 swaps per erroneous query), the MRR for U decreases by only 8% (from 0.484 to 0.443) while the MRR for C:D decreases by 33% (from 0.379 to 0.253); as a result, U's MRR outperforms other approaches by at least 75% ($p < 0.01$). In fact, Figures 4e-f show that C:D can become significantly worse than C+D. Meanwhile, even if 100% U queries have two pairs of directory path and content terms switched, U still outperforms C+D by a wide margin.

Further, these results show that node inversion is critically important to U's accuracy when queries contain ordering mistakes. U always outperforms U-no-NI, with the difference in accuracy increasing significantly as queries contain more inaccuracies. In fact, when queries contain 2 swap mistakes, U-no-NI performs worse than C+D. This is because U-no-NI has to rely on node deletion to eliminate mis-ordered terms, thereby loosing the terms' differentiation power for ranking answers.

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. It is also robust against labeling inaccuracies (i.e., structure terms identified as content and vice versa) in query conditions.

5.5 Comparing with Existing Search Tools

We have also compared our approach with GDS and TopX [21]. We choose GDS because of its popularity and the fact that it ranks returned results. Many other existing search tools do not; for example, Spotlight only returns a single ranked result (the top ranked), organizing the remaining results using attributes such as dates and file types. We choose TopX because it implements a related search approach, although it was designed for XML retrieval.

GDS matches keywords against both pathnames and content. Thus, we use the set of queries constructed for C+D from Section 5.1. The results for GDS Windows version 5.9 are similar to C+D in Figure 4 for ranks 1-6, degrading significantly after.

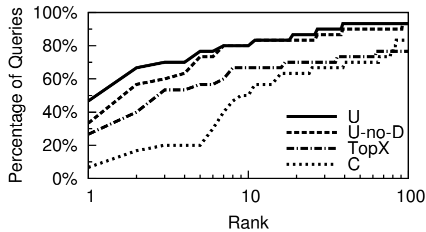


Figure 5: The CDFs of ranks of target files for 30 email and ebook queries.

Table 3: Average precision for $k = 5, 10$.

Search Method	Precision, k=5	Precision, k=10
U	0.52	0.32
U-no-D	0.52	0.32
TopX	0.44	0.28
GDS	0.32	0.18
C	0.28	0.14

TopX treats each document as a tree of nodes, each with a tag and some content. Each internal node's content is the concatenation of the contents of its children. TopX adapts the Okapi BM25 scoring model to score content. It transitively expands all structural dependencies and counts the number of conditions matched by a file to score structure. It combines the two scores using summation with an emphasis on the importance of structure query conditions.

We use the open source TopX package with default settings. We limit our comparison to the 30 email and ebook queries from the query set used in Figure 4a since TopX only supports XML and text files (emails and ebooks are stored in XML format). Directory terms are dropped from the queries since TopX only considers internal structure.

Figure 5 plots the CDFs of ranks of target files for the query set. We observe that TopX outperforms content-only search (C). However, our unified approach (U) outperforms TopX, even if we do not use directory information (U-no-D). Further, while not shown here, TopX's accuracy drops rapidly in the presence of ordering errors because it is much less flexible than U in dealing with them. For example, TopX does not allow structural parts of queries to be matched with content and vice versa.

To compare using a more conventional IR metric, Table 3 shows the average precision for a subset of five randomly selected queries. We manually examined the top k ranked files returned for each query and marked them relevant or not. (We cannot measure recall because this requires evaluating the relevance of all files against all queries.) Our approach again outperforms content-only search, GDS, and TopX.

5.6 Query Processing Performance

Figure 6 plots CDFs of query processing times for the query set considered in Section 5.4 for different values of k. (Processing times for the additional query sets mentioned in Section 5.4 and queries in Section 5.3 are similar.) 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, for k=10, approximately 70% of the queries still complete in less than 4 seconds, with the longest requiring 17 seconds. Further, query processing scales well with k, degrading only slightly with increasing k.

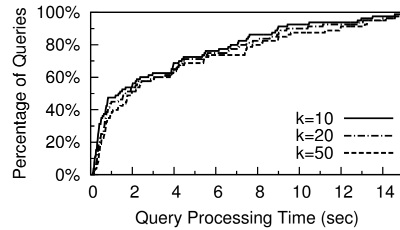


Figure 6: CDFs of query processing times to find and rank the top k relevant files for queries considered in Section 5.4.

There are several possible performance optimizations. First, U is heavily penalized when processing high frequency terms such as *subject* because of its path matching algorithm. This penalty largely arises from our naive use of the Berkeley DB and can be significantly reduced with some coding effort. Second, there are opportunities for skipping unnecessary computation that we have not implemented.

5.7 Storage Cost

Our indexes require 1.9 GB of persistent storage, which is 11% of the data set size (16.6 GB). Lucene requires 676 MB of storage to index the same data set. While this is almost a three-fold increase, the total required storage is still quite reasonable. Further, it makes sense to trade-off a small amount of disk space 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.

A number of efforts have explored structure and content search for XML [1, 2, 3, 8, 14]. However, these approaches have either ignored content [1], considered content nodes as atomic values that could be part of the structural relaxation but whose terms could not be searched individually [2], or considered content and structure separately [3, 8, 14]. The latter works [3, 8, 14] propose techniques that use structure conditions as templates on which to apply the content searches; the quality of content matches is penalized when the structural match is not perfect. Furthermore, [8, 14] compute indexes around a subset of predefined XPath fragments (structure). In contrast, our framework unifies scoring for structure and content, allowing it to be robust to mistakes resulting from users mistakenly believing content terms are in the structure and vice versa. Our framework also handles arbitrary DAG structures and unifies external structure and internal structure, removing the need for users to know the underlying physical organization of data (which can be application-dependent) when composing queries.

Our previous work [19] only considers external structure

(e.g., directories) and scores structure and content separately, applying a combining function to assign a final score to data matches. In contrast, the current work unifies the matching and scoring of external structure, internal structure, and content, bringing the advantages already mentioned above. In addition, the current work views the entire file system as a unified data tree, allowing searches that cross file boundaries in a directory-based file system. We compared the two systems using the query sets from Section 5.4. We do not show the results, however, because our previous system performs similar to C:D (slightly better) for this data set, given that the namespace is not large and so using structural relationships only gives a slight advantage over using terms from directory names.

A further difference between our work and the previous works in XML mentioned above is the use of node inversion. This relaxation was introduced in [19] but was only applied to the structure query conditions. The current work allows for node inversion between structure and content query conditions, which is the key to tolerating mistakes that interchange the two. However, allowing node inversion means that we cannot apply common query processing techniques (mostly derived from [6]) that rely on the query having a rigid tree structure; our query structure may contain node groups representing possible permutations. Consequently, Algorithms 1 and 2 are non-trivial extensions of *PathStack* [6] that had to be developed.

Partial path queries [23] also extended *PathStack*, allowing fuzziness in the query conditions by keeping some structural relationships between query nodes undefined. Our work supports more complex path queries containing term permutations introduced by node inversion.

Ranking techniques based on learning and query selectiveness are proposed in [9]. Their ranking formula uses a linear function to aggregate weights for various file features such as name, size, and creation date.

7. CONCLUSIONS 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 important research directions for efficient and high-quality search tools.

In this paper, we have focused on files as the result unit. In the future, we will relax this restriction to allow for logical units of data to be returned. For example, a set of photos taken at the same time and location or an email thread could constitute and be returned as a single logical entity. Alternatively, a search might return just a section in a file.

Acknowledgements. We thank the reviewers and our shepherd, Fabio Casati, for helping us improve the paper.

8. REFERENCES

[1] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree Pattern Relaxation. In *Proc. of the EDBT Conference*, 2002.

[2] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman. Structure and Content Scoring for XML. In *Proc. of the VLDB Conference*, 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 SIGMOD Conference*, 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 IUI Conference*, 2007.

[6] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. of the SIGMOD Conference*, 2002.

[7] Y. Cai, X. L. Dong, A. Halevy, J. M. Liu, and J. Madhavan. Personal Information Management with SEMEX. In *Proc. of the SIGMOD Conference*, 2005.

[8] D. Carmel, Y. S. Maarek, M. Mandelbrod, Y. Mass, and A. Soffer. Searching XML Documents via XML Fragments. In *Proc. of the SIGIR Conference*, 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 VLDB Conference*, 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 SIGMOD Conference*, 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 CIDR Conference*, 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 EDBT Conference*, 2008.

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

[21] 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.

[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, 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 *Proc. of the WWW Conference*, 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.