ISSN No. 0976-5697

Volume 5, No. 2, March 2014 (Special Issue)

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

A System for Personal Information Management using an Efficient Multidimensional Fuzzy Search

A.Seenu
Department of Computer Science and Engineering
SVECW Bhimavaram, India
cnuaaluri@gmail.com

M.R.Narasinga Rao
Department of Computer Science and Engineering
K L University India
Ramanarasingarao@kluniversity

U.S.S.Padma Jyothi
Department of Computer Science and Engineering
SVECW Bhimavaram, India
padmajyothi64@gmail.com

Abstract: Users access a large amount data and store in personal information management systems, There is a need to retrieve disparate data in a simple and efficient way using some search tools. IR-style ranking is supported by existing tools, but structure(path of a file) and metadata(includes name,author,title,date) are considered as filtering conditions in this approach. Normal search is performed based on keyword conditions but the multidimensional search allows search using structure and metadata. It integrates the three dimension score into unified score where each score belongs to a dimension. In order to identify the relevant files which matches with the given query we make use of indexes and algorithms. We can improve ranking accuracy by performing the experiments for our approach. By using query processing strategies , the fuzzy search approach can be used in every day of our life.

Keywords: Multidimensional search, personal information management system, query processing

I. INTRODUCTION

The data which is to be stored in personal information management systems is rapidly increasing, This explosion of information needs a powerful search tools to access often very disparate data in a simple and efficient manner. Such tools should provide both high-quality scoring mechanisms and efficient query processing capabilities.

Inorder to perform keyword search and locate personal information stored in file there are numerous search tools such as the commercial tools Google Desktop Search [1] and Spotlight [19]. However, these tools usually support textual part of the query for searching a file—similar to what has been done in the Information Retrieval (IR) community,but only consider structure (e.g., file path) and metadata (e.g.,filetype, title, author) as filtering conditions.

Recently, the research community has turned its focus on search over to Personal Information and Dataspaces [10], [12], [14], which consist of different data collections. However, in this search tools, these works focus on IR-style keyword queries and use other system information only to guide the keyword-based search.

Keyword only searches are often insufficient, as illustrated by the following example: Consider a user stores his personal information in the file system of a personal computing device. In addition to the actual file content, location information (e.g., directory structure) and a metadata information (e.g., access time, date) are also stored by the file system.

In such a scenario, the user might want to ask the query:
[file type = *.pdf AND accesstime = 12/03/2014 AND content = “Fuzzy search” AND structure = /docs/find/search]

Current tools would answer this query by returning all files of type *.pdf created on 12/03/2014 under the directory /docs/find/search (as filtering conditions) that have content similar to “Fuzzy search” (ranking expression), ranked based on how close the content matches “Fuzzy search” using some underlying text scoring mechanism. Because all information except the content are used as filtering conditions, files that are very relevant to the query, but which do not satisfy these exact conditions would be ignored. For example, *.txt docu-ments created on 12/03/2014 and files in the directory /arc/search/fuzzy containing the terms “Fuzzy Search” would not be returned.

We argue that allowing flexible conditions on structure and metadata can significantly increase the quality and usefulness of search results in many search scenarios. For instance, in Eg 1, the user might not remember the exact creation date of the file but remembers that it was created around 21/03/2014. Similarly, the user might be interested in files of type *.pdf but might also want to consider relevant files of different but related types (e.g., *.tex or *.txt). Finally, the user might misremember the directory path under which the file was stored. In this case, by using the date, size, title and structure conditions not as filtering conditions but as part of the ranking conditions of the query.

The challenge is then to score answers by taking not only textual components alone but together with flexibility in the structural and metadata components of the query. Efficient algorithms are need to identify the best query results, without considering all the information in the file system.

We propose a novel approach that allows users to perform fuzzy searches across three different dimensions: structure content, and metadata. Then for each dimension an IDF based scoring is provided and a unified scoring

CONFERENCE PAPER
Two day National Conference on Advanced Trends and Challenges in Computer Science and Applications
Organized by: Shree Vishnu Engineering College for Women, Bhimavaram A.P.
Schedule: 18-19 March 2014

framework for multi-dimensional queries over personal information file-systems. In this we present new data structures and index construction optimizations to make finding and scoring fuzzy matches efficient.

Our work could be extended to a variety of dataspace applications and queries, and this paper focus on a file search. we consider the granularity of the search results to be a single file in the personal information system. Of course, our techniques could be extended to a more flexible query model where pieces of data within files could be returned as results.

II. UNIFIED MULTI-DIMENSIONAL SCORING

In this section, we present our unified framework we can assign the scores to the files for closely mapped query condition of different query dimnesions. There are three scoring dimensions: content defines the textual content of the files, metadata defines the system information related to the files, and structure defines the directory path to access the file.

We represent files and their associated metadata and structure information as XML documents. Metadata and structure conditions in addition to keyword-based content conditions are explained by using simplied version of XQuery

Any file that is related to one or more of the query conditions is correct answer for the query.Each file will be assigned a score for each dimension on how close it matches the corresponding query condition. Scores across multiple dimensions are filtered into a single overall score for ranking of answers.

Our scoring strategy is based on an IDF based interpretation of scores. For each query condition, we rank files based on the least relaxed form of the condition that each file matches. Scoring along all dimensions is uniformly IDF based which helps us to meaningfully combine multiple single-dimensional scores into a unified multi-dimensional score.

A. *Scoring Content:*

We use standard IR relaxation and scoring techniques for content query conditions. Specifically, we adopt the TF·IDF scoring formulas from Lucene [6], a state-of-the-art keyword search tool. These formulas are as follows:

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

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

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

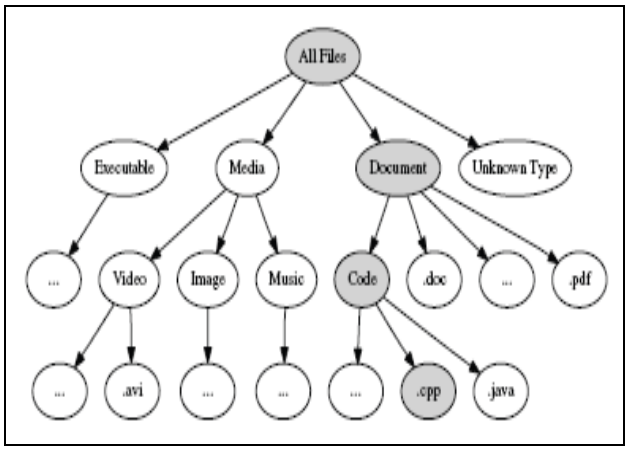

Figure:1

Where Q is the content query condition, f is the file score given to a file, N is the total sum of files, N is the number of files containing the term t, and NormLength(f) is a normalizing factor that is a function of f 's length. 2 Note that relaxation is an integral part of the above formulas since they score all files that contain a subset of the terms in the query condition.

B. *Score of Metadata:*

We introduce a hierarchical relaxation approach for each type of searchable metadata to support scoring. For example, Figure 1 shows the relaxation levels for file types, represented as a Direct Acyclic Graph3. Each leaf node of graph represents a specific file type (e.g., pdf files). Each internal node represents a more general file type that is the union of the types of its children (e.g., Media is the union of Video, Image, and Music) and thus is a relaxation of its descendants. The set of files matching a node must be equal to or sub-sum of files matching each of its children nodes. This enable that the score of a file matching a more relaxed form of a query condition is always less than or equal to the score of a file matching a less than relaxed form (see Equation 4 below).

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

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

$$score_{Meta}(Q,f) = \frac{\log\left(\frac{N}{nFiles(commonAnc(n_M,n_f))}\right)}{\log(N)}$$

Where N is the total sum of files, nM is the deepest node that matches M , nf is the deepest DAG node that matches f , commonAnc(x, y) returns the closest common ancestor of nodes x and y in the relaxation hierarchy, and nFiles(x) returns the number of files that match node x. The score is normalized by log(N) so that a single perfect match would have the highest possible score of 1.

C. Score of a Structure:

To organize their files most of users use a hierarchical directory structure. When searching for a particular file, a user may often remember only some components of the containing directory path and approximate ordering than the exact path itself. Thus, allowing for some approximation on structure query conditions is required because it allows users to leverage their partial memory to help the search engine locate the required file.

Our structure scoring strategy extends initial work on XML structural query relaxations [4], [5]. Specifically, the node inversion relaxation introduced here is novel and introduced to handle possible mis-ordering of pathname components when specifying structure query conditions in personal file systems. Assuming that structure query conditions are given as non-cyclic paths (i.e., path queries), these relaxations are:

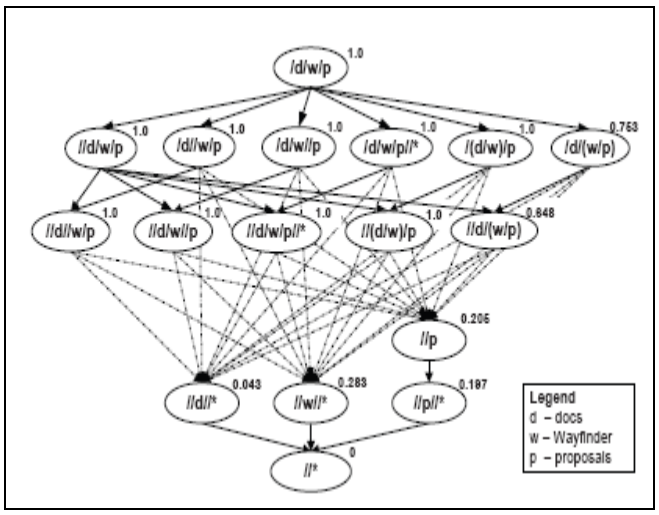

Figure 2

Generalization of an Edge is used to relax a parent-child relationship to an ancestor-descendant relationship. For example, applying edge generalization to /c/d would result in /c//d.

Path Extension is used to extend a path P such that all files within the directory subtree rooted at P can be considered as result. For example, applying path extension to /c/d would result in /c/d//∗.

Inversion Node is used to permute nodes within a path query P. To represent possible permutations, we introduce the notion of node group as a path where the placement of edges are fixed and (labeled) nodes may permute. Permutations can be applied to any adjacent nodes or node groups which have an exception for the root and *nodes. A permutation combines adjacent nodes, or node groups, into a single node group while preserving the relative order of edges in P . For example, applying node inversion on c and d from /c/d/e would result in /c/(d/e), allowing for both the original query condition as well as /a/c/b. The (b/c) part of the relaxed condition /c/(d/e) is called a node group.

Deletion Node is used to delete a node from a path. Node deletion can be applied to any path query or node group but cannot be used to drop the root node or the ∗ node.

To delete a node n1 in a path query P:

a. If n1 is a leaf node, n1 is deleted from P and P – n1 is extended with //∗. This is to ensure containment of the exact answers to P in the set of answers to P', and monotonicity of scores.
b. If n is an internal node, n1 is dropped from P and parent(n) and child(n) are connected in P with //. For example, deleting node c from c/d/e results in c/d//∗ because c/d//∗ is the most specific relaxed path query containing a/b/c that does not contain c. Similarly, deleting c from c/d/e//∗ results in c//d//∗.To delete a node n1 that is within a node group N in a path query P , the following steps are required to ensure answer containment and monotonicity of scores:
c. n1 and one of its adjacent edge in N are dropped from N . Every edge within N becomes an ancestor-descendant edge. If n1 is the only node left in N , N is replaced by that node in P .
d. Within P the surrounding edges of N are replaced by ancestor-descendant edges.

$$score_{Struct}(Q,f) = \max_{p \in R(Q)} \{score_{idf}(p) \mid f \in F(d,p)\},$$

$$score_{idf}(p) = \frac{\log\left(\frac{N}{N_p}\right)}{\log(N)}, \quad N_p = |F(d,p)|$$

e. If N is a leaf node group, the result query is extended with //∗.

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

D. Score Aggregation:

We cumulate the above single-dimensional scores into a filtered multi-dimensional score to provide a fused ranking of files relevant to a multi-dimensional query. To do this, we construct a query vector,

V_Q having a value of 1 (exact match) for each dimension and a file vector, V_F, consisting of the single-dimensional scores of file F with respect to query Q. (Scores for the content dimension is normalized against the highest score for that query condition to get values in the range [0, 1].) We then compute the projection of V_F onto V_Q and the length of the resulting vector is used as the cumulative score of file F . In its current form, this is simply a linear combination of the component scores with equal weighting. The vector projection method, however, provides a framework for future investigation of more complex aggregations.

III. QUERY PROCESSING

We adapt an existing algorithm called the Threshold Algorithm [13] to make query processing. Threshold Algorithm uses a threshold condition which avoids evaluating the possible matches to a query, focusing on identifying the k best answers. It takes several sorted lists as a input , each containing the system's objects such as files (in our scenario) sorted in descending order according to their relevance scores for a particular attribute as dimension (in our scenario), and dynamically they can accesses sorted lists until the threshold condition is met to find the k best answers.

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

We will now present these indexing structures and algorithms.

A. Evaluation of Content Scores:

As mentioned in Section 2.1, we use existing TF·IDF methods to score the content dimension. By using Random accesses we can find the term frequency in the entire file system as well as in a particular file. we keep the file in a sorted order according to their Term-Frequency scores, normalized by file size, for that term. 4 We then use the Threshold Algorithm recursively to return files in sorted manner according to their content scores for queries that having more than one term.

B. Evaluation of Metadata Scores:

Sorted access for a metadata condition is implemented using the appropriate relaxation DirectAcyclicGraph index. First, exact matches are identified by identifying the deepest Direct Acyclic Graph node that matches the given metadata condition (see Section 2.2). Once all exact matches have been retrieved from N's leaf descendants, approximate matches are produced by traversing up the Direct Acyclic Graph to consider more approximate matches. Each parent contains a bigger range of values than its child nodes, which ensures the matched nodes are returned in decreasing order of metadata scores. For content dimension, we can use the Threshold Algorithm recursively to return files in sorted order for queries that contain multiple metadata conditions.

Random accesses for a metadata condition require positioning in the appropriate Direct Acyclic Graph index the closest common ancestor of the deepest node that matches the condition and the deepest node that matches the file's metadata attribute (see Section 2.2). This is implemented as an efficient Direct Acyclic Graph traversal algorithm.

IV. EVALUATION OF STRUCTURE SCORES

The structure score of a file for a query condition depends on how close the directory in which the file is associated

To compute the structure score of a file f in a directory d1 that matches the (exact or relaxed) structure condition P of a given query, we have to determine all the directory paths, including d1 that match P. We will then sum the number of files contained in all the directories matching P to compute the structure score of these files for the query using Equation 6. The score computation step is straightforward; the complexity resides in the directory matching step. Node inversions complicate matching query paths with different directories, as required possible permutations have to be measured. Particular techniques and their supporting index structures required to be developed.

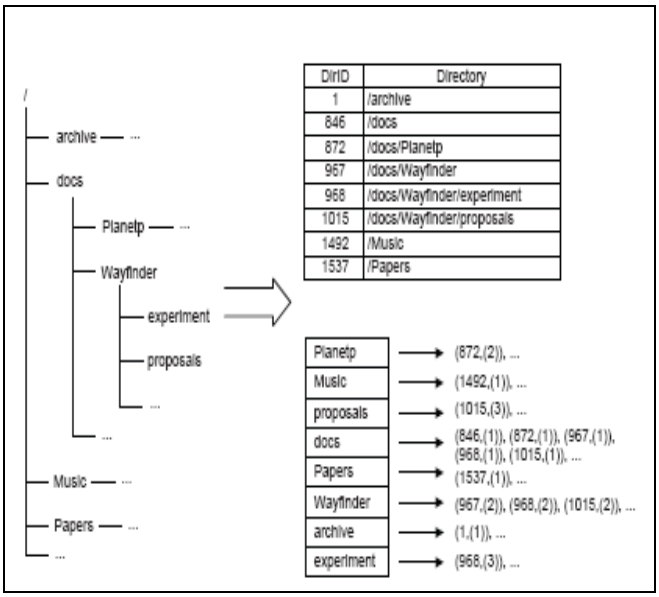

Figure: 3

We use a two-phase algorithm to identify all the required directories that match a query path. First, we identify a set of candidate directories using the observation that for a directory d1 to match a query path P , it is necessary for all the components in P to appear in d1. For example, the directory/docs/proposals/final/Wayfinder is a potential match for the query path /docs/(Wayfinder//proposals) since the directory contains all three components docs, Wayfinder, and proposals. We implement an inverted index mapping components to directories to support this step (see Figure 3).

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

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

lationships extracted for the query path and thus concludes that the candidate directory is a valid match for the query path. For a path query P with |P|components, our query matching algorithm has worst-case I/O complexity linear in the sum of sizes of the |P | inverted lists, and CPU time complexity linear in the length of the smallest of the |P | inverted lists. The worst-case space complexity does not exceed the maximum length of directory pathnames. Details of the algorithm and complexity analysis can be found in [29].

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

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

V. OPTIMIZING QUERY PROCESSING IN THE STRUCTURE DIMENSION

In this section, we present our dynamic indexes and algorithms for efficient processing of query conditions in the structure dimension. This dimension brings the following challenges:

a. The Direct Acyclic Graphs representing relaxations of structure conditions [4], [17] are query-dependent and so have to be built at query processing time. However, since these DAGs grow exponentially with query size, i.e., the number of components in the query, efficient index building and traversal techniques are critical issues.
b. The Threshold Algorithm requires efficient sorted and random access to the single-dimension scores (Section 3).

We propose the following techniques and algorithms to address the above demands. We incrementally build the query dependent Direct Acyclic Graph structures at query time, only materializing those Direct Acyclic Graph nodes necessary to reply a query (Section 4.1). To improve sorted access efficiency, we propose techniques to skip the scoring of unnecessary Direct Acyclic Graph nodes by taking advantage of the containment property of the Direct Acyclic Graph (Section 4.2). We improve random accesses using a novel algorithm that efficiently locates and evaluates only the parts of the Direct Acyclic Graph that match the file requested by each random access (Section 4.3).

A. *Incremental Identification of Relaxed Matches:*

As mentioned in Section 2.3, we represent all possible relaxations of a query condition and corresponding IDF scores using a Direct Acyclic Graph structure. Scoring an entire query relaxation Direct Acyclic Graph can be expensive as they grow exponentially with the size of the query condition. For example, there are 5, 21, 94, 427, and 1946 nodes in the respective com-

plete Direct Acyclic Graph for query conditions /a, /a/b, /a/b/c, /a/b/c/d, /a/b/c/d/e. However, in many cases, enough query matches will be found near the top of the DAG, and a large portion of the Direct Acyclic Graph will not need to be scored. Thus, we use a lazy evaluation approach to incrementally build the DAG, expanding and scoring DAG nodes to produce additional matches when needed in a greedy fashion [29]. The partial evaluation should nevertheless ensures that directories (and therefore files) are returned in the order of their scores.

For a simple top-k evaluation on the structure condition, our lazy Direct Acyclic Graph building algorithm is applied and stops when k matches are identified. For complex queries involving multiple dimensions, the algorithm can be used.Random accesses are more problematic as they may access any node in the DAG. The Direct Acyclic Graph building algorithm can be used for random access, but any random access may lead to the materialization and scoring of a large part of the DAG.5

B. *Improving Sorted Accesses:*

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

Not every possible relaxation leads to the discovery of new matches. For example, in Fig2, the query paths /docs/Wayfinder/proposals, //docs/Wayfinder/proposals, and //docs//Wayfinder/proposals have exactly the same scores of 1, which means that no additional files were retrieved after relaxing /docs/Wayfinder/proposals to either //docs/Wayfinder/proposals or //docs//Wayfinder/proposals (Equation 6). By extension, if two DAG nodes share the same score, then all the nodes in the paths between the two DAG nodes must share the same score as well per the DAG definition. This is formalized in Theorem 1

Theorem 1: Given the structural $score_{idf}$ function defined in Eq 6, if a query path P' is a relaxed version of another query path P, and $scoreidf(P') = score_{idf}(P)$ in the structure DAG, any node P'' on any path from P to P' has the same structure score as $score_{idf}(P)$, and $F(P') = F(P'') = F(P)$, where F(P) is the set of files matching query path P.

Proof: (Sketch) If $score_{idf}(P') = score_{idf}(P)$, then by definition $NP' = NP$ (Equation 6). Because of the containment condition, for any node P'' on any path from P to P', we have $F(P')$ or $F(P'')$ or F(P) and $NP' \geq NP'' \geq NP$. Thus $NP' = NP'' = NP$ and $F(P') = F(P'') = F(P)$, since otherwise there exists at least one file which belongs to $F(P')$ (or $F(P'')$) but does not belongs to F(P) and $NP'' = NP$ (or NP'''=NP), contradicting our assumption $NP' = NP$ (and $NP'' = NP$).

Theorem 1 can be used to speed up sorted access processing on the DAG by skipping the score evaluation of DAG nodes that will not contribute to the answer, since the score evaluation of DAG nodes can be expensive. We propose Algorithm 1, DAG-Jump.

It includes two steps: (a) starting at a node corresponding to a query path P, the algorithm performs a depth-first traversal and scoring of the DAG until it finds a

parentchild pair, P' and child(P'), where $score_{idf}$ (child(P')) < $score_{idf}$ (P); and (b) score each node P'' at the same

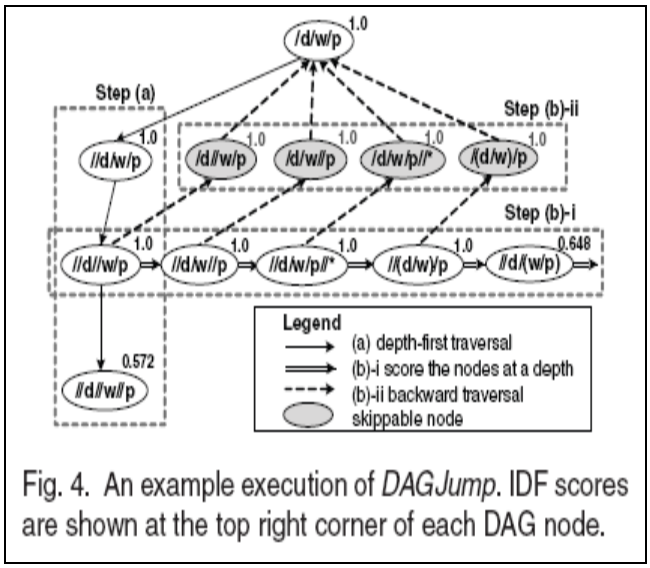

Fig. 4. An example execution of *DAGJump*. IDF scores are shown at the top right corner of each DAG node.

Depth (distance from the root) as P' if $score_{idf}$ (P'') =$score_{idf}$ (P), then traverse all paths from P'' back toward the root; on each path, it can reach a previously scored node P *, where $score_{idf}$ (P*) = $score_{idf}$ (P); all nodes on all paths from P'' to P* can be dropped since they have the same score as P''.

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

Algorithm 1. DAG-JUMP(srcNode)

```
a.  S ← getScore(srcNode)
b.  currentNode ← srcNode
c.  loop
d.      targetDepth ← getDepth(currentNode)
e.      childNode ← firstChild(currentNode)
f.      If getScore(childNode) ≠ s or
g.          hasNoChildNodes(childNode) then
h.          exit loop
i.      currentNode ← childNode
j.  for each n getDepth(n) = targetDepth and
        getScore(n) = s do
      Evaluate bottom-up from n and identify ancestor
      node set S s.t. getScore(m)=s, ∀m€ S
k.  for each m€ S do
l.      for each n' on path p € getPaths(n,m) do
m.          setScore(n',s)
n.          setSkippable(n')
o.          if notSkippable(m)  then
p.              setSkippable(m)
```

C. Improving Random Accesses:

Random accesses is required for top-k query processing in DAG. Using sorted access to emulate random access tends to be very inefficient as it is likely the top-k algorithm will access a file that is in a directory that only matches a very relaxed version of the structure condition.

While the DAG-Jump algorithm somewhat alleviates this problem by reducing the number of nodes that need to be scored, efficient random access remains a critical problem for efficient top-k evaluations. We present the RandomDAG algorithm to optimize random accesses over our structure DAG. The key idea behind RandomDAG is to skip to a node P in the DAG that is either a close ancestor of the actual least relaxed node P' that matches the random access file's parent (containing) directory d or P' itself and only materialize and score the sub-DAG rooted at P as necessary to score P'.

The intuition is that we can identify P by comparing d and the original query condition. In particular, we compute the intersection between the query condition's components and d. P is then computed by dropping all components in the query condition that is not in the intersection, replacing parent-child with ancestor-descendant relationships as necessary. The computed P is then guaranteed to be equal to or an ancestor of P'. As DAG nodes are scored, the score together with matching directories are cached to speed up future random accesses.

Algorithm 2 Random-DAG (root, DAG, F)

```
a.  P ← getDirPath(F)
b.  if  p € DAGCache then
c.      return getScoreFromCache(DAGCache ,p)
d.  droppedComponents ←
        extractComponents(root) -
    extractComponents(p)
e.  p'←root
f.  for each component  € droppedComponents do
g.    p' ← nodeDeletion(p',component)
h.  loop
i.      n ← getNextNodeFromDAG(p')
        {getNextNodeFromDAG incrementally build
    a
        sub-DAG rooted at p' and  returns the next
    DAG
        node in decreasing order of score.}
j.   fileMatches ← matchDirectory(getQuery(n))
k.   dirPaths ← getDirpaths(fileMatches)
l.   addToCache(DAGCache,dirPaths,getScore(n))
m.  if p € dirPaths  then
n.    return getScore(n)
```

As an example, for our query condition /docs/Wayfinder/proposals in Figure 2, if the top-k algorithm wants to perform a random access to evaluate the structure score of a file that is in the directory /archive/proposals/Planetp, RandomDAG will first compute the close ancestor to the node that matches /archive/proposals/Planetp as the intersection between the query condition and the file directory, i.e., //proposals, and will jump to the sub-DAG rooted at this node. The file's directory does not match this query path, but does match its child //proposals//* with a structure score of 0.197. This is illustrated in Figure 5 which shows the parts of the DAG from Figure 2 that would need to be accessed for a random access to the score of a file that is in the directory /archive/proposals/Planetp.

CONFERENCE PAPER
Two day National Conference on Advanced Trends and Challenges in Computer Science and Applications
Organized by: Shree Vishnu Engineering College for Women, Bhimavaram A.P.
Schedule: 18-19 March 2014

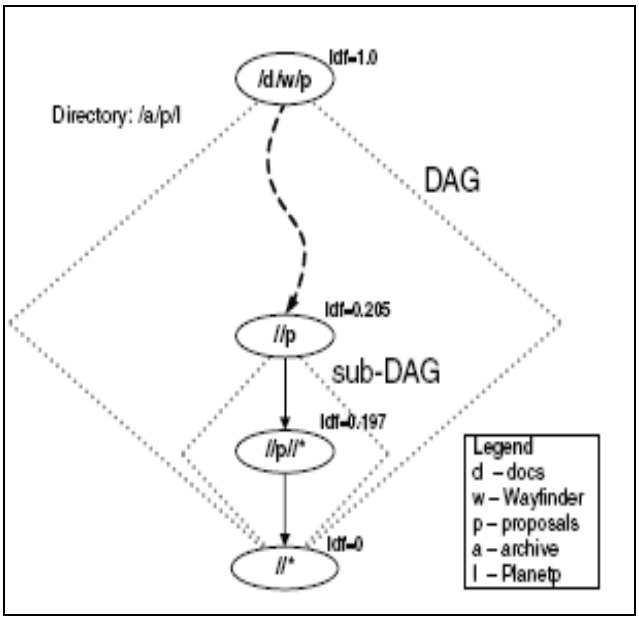

Figure: 5

VI. EXPERIMENTAL RESULTS

We now experimentally evaluate the potential for our multidimensional fuzzy search approach to improve ranking accuracy. We also report on search performance achievable using our index structures, scoring algorithms, and also top-*k* adaptation.

A. Experimental Setup:

a. Experimental environment: All experiments were performed using a prototype system implemented in Java. We use the Berkeley DB [18] to persistently store all indexes and Lucene to rank content. Experiments were run on a PC with a 64-bit hyper-threaded 2.8 GHz Intel Xeon processor, 2 GB of memory, and a 10K RPM 70 GB SCSI disk, running the Linux 2.6.16 kernel and Sun's Java 1.4.2 JVM. Reported query processing times are cummalation of 40 runs, after 40 warm up runs to avoid measurement JIT effects. All caches (except for any Berkeley DB internal caches) are flushed at the beginning of each run.

b. Data set: As noted in [12], there is a large amount data sets and benchmarks to evaluate search over personal information management systems. Then a data set contains files and directories from the working environment of one of the authors. This data set contains 14.3 GB of data from 24,926 files organized in 2,338 directories; 24% of the files are multi-media files (e.g., music and pictures), 17% document files (e.g., pdf, text, and MS Office), 14% email messages,7 and 12% source code files. The average depth of a directory was 3.4 with the longest being 9. On average, each directory contains 11.6 subdirectories and files. The system extracted 347,448 unique content terms. File modification dates span 10 years. 75% of the files are smaller than 177 KB.

B. Impact of Flexible Multi-Dimensional Search:

To improve ranking accuracy using two example search scenarios. In each scenario, we initially construct a content only query intended to retrieve a specific target file and then expand that particular query is extended to other dimensions. For each query, we consider the status of the aimed file by our approach together with whether the target file would be ranked at all by today's typical filtering approaches on non-content query conditions. An example of results in Table 1. In the first example, the target file is the novel "Time Machine" by H.G.Well, located in the directory path */Personal/Ebooks/Novels/*, and the set of query content terms in our initial content-only query Q1 contains the two terms *time* and *machine*.

While the query is quite reasonable, the terms are common enough that they appear in many files, leading to a ranking of 18 for the target file.Q2 augments Q1 with the exact matching values for file type, modification date, and containing directory. This helps in ranking the target file to 1. The left-s over queries look at what happens when we provide an incorrect value for the non-content dimensions. For example, in query Q10, a group of correct but wrongly ordered components in the directory name still brings the ranking up to 1. In comparisons, if such directories were given as filtered results, the target file would be considered mismatch to the query and not ranked at all; queries which contain a "*" next to our technique's rank result represent those in which the target file would not be considered as a relevant answer given today's typical filtering approach.

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

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

Query Evaluation Results

Query	Content	Query Conditions: Type	Modification Date	Structure	Rank	Comments on Relaxation from Query Q1/Q14
Search Scenario 1: The user searches for the electronic book "The Time Machine". Target file: /Personal/Ebooks/Novels/The Time Machine.pdf Structure condition: /Personal/Ebooks/Novels (abbreviated as /p/e/n, 'c' in the incorrect path stands for 'Comics')						
Q1	time, machine	-	-	-	18	Base Query
Q2	time, machine	.pdf	22 Jan 07 18:09	/p/e/n	1	Correct Values (all dimensions)
Q3	time, machine	.pdf	-	-	9	Correct File Type
Q4	time, machine	.doc	-	-	42 *	Incorrect File Type
Q5	time, machine	Docs.	-	-	18	Relaxed Range
Q6	time, machine	-	21-27 Jan 07	-	1	Relaxed Range (Week of month)
Q7	time, machine	-	23 Jan 07 18:09	-	1 *	Incorrect Date (off by 1 day)
Q8	time, machine	-	15 Feb 07 18:09	-	4 *	Incorrect Date (off by 1 month)
Q9	time, machine	-	-	/p/e/n	1	Correct Path
Q10	time, machine	-	-	/m/e	1 *	Incorrect Order/Correct Components
Q11	time, machine	-	-	/p/e/e	2 *	Incorrect Path
Q12	time, machine	Docs.	Jan 07	/p/e	1	Relaxed Range (all dimensions)
Q13	time, machine	.pdf	15 Feb 07 18:09	/c/e	1 *	Incorrect Date and Path
Search Scenario 2: The user searches for an email that discusses the java implementation of the IR algorithm for the file system Wayfinder. Target file: /Personal/Mail/Code/Java/920.SA_____2006101819215778.xml Structure condition: /Mail/Code/Java (abbreviated as /m/c/j, 'p' in the incorrect path stands for 'Python')						
Q14	Wayfinder, IR	-	-	-	42	Base Query
Q15	Wayfinder, IR	.xml	18 Oct 06 14:21	/m/c/j	1	Correct Values (all dimensions)
Q16	Wayfinder, IR	.xml	-	-	35	Correct File Type
Q17	Wayfinder, IR	.txt	-	-	39 *	Incorrect File Type
Q18	Wayfinder, IR	Docs.	-	-	39	Relaxed Range
Q19	Wayfinder, IR	-	15-21 Oct 06	-	1	Relaxed Range (Week of month)
Q20	Wayfinder, IR	-	17 Oct 06 14:21	-	1 *	Incorrect Date (off by 1 day)
Q21	Wayfinder, IR	-	18 Nov 06 14:21	-	8 *	Incorrect Date (off by 1 month)
Q22	Wayfinder, IR	-	-	/m/c/j	1	Correct Path
Q23	Wayfinder, IR	-	-	/j/m	1 *	Incorrect Order/Correct Components
Q24	Wayfinder, IR	-	-	/m/c/p	1 *	Incorrect Path
Q25	Wayfinder, IR	Docs.	Oct 06	/m/c	1	Relaxed Range (all dimensions)
Q26	Wayfinder, IR	.txt	18 Nov 06 14:21	/j/c	1 *	Incorrect Date and Path

TABLE 1

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

C. Storage Cost:

We report the cumulative size of our static indexes of Section 3 to show that our approach is practical with respect to both space (storage cost) and time (query processing performance). In total, our indexes require 246 MB of storage, which is less than 2% of the data set size (14.3GB). This storage is dominated by the content index, which accounts for almost 92% of the 246 MB. The indexes are so compact compared to the data set because of the large sound (music) and video (movie) files. As future data sets will be increasingly media rich, we expect that our indexes will continue to require a relatively insignificant amount of storage.

VII. CONCLUSION

Multidimensional queries make use of the scoring framework for personal information management systems. Metadata and structure relaxations are defined and we proposed structure,metadata and content query conditions using a approach IDF based scoring . By using this unified scores we can aggregate the scores easily. In order to support efficient multidimensional queries we have designed query processing optimizations, indexing structures and construction.

The scoring framework and query processing techniques are implemented and evaluated. By using this evaluation the aggregation multidimensional score approach preserves the properties of individual dimension scores and ranking accuracy has been increased significantly. We can make the multidimensional search efficient for daily usage by making use of our indexes and optimizations, which results in good query performance.

VIII.REFRENCES

[1]. Google desktop. http://desktop.google.com

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

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

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

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

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

[7]. R. A. Baeza-Yates and M. P. Consens. The continued saga of DB-IR integration. In Proc. of the Intl. Conference on Very Large Databases (VLDB), 2004.

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

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

[10]. Y. Cai, X. L. Dong, A. Halevy, J. M. Liu, and J. Madhavan. Personal Information Management with SEMEX. In Proc. of the ACM Intl. Conference on Management of Data (SIGMOD), 2005.

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

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

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

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

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

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

[17]. S. Amer-Yahia, P. Case, T. R¨olleke, J. Shanmugasundaram, and G. Weikum. Report on the DB/IR panel at SIGMOD 2005. SIGMOD Record, 34(4), 2005.

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

[19]. 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.