Interscience Management Review

Volume 3 | Issue 2 Article 7

July 2010

Efficient Multi Dimensional Nodal Fuzzy Search for Information Management System

M Raja Narayana
Madanapalle Institute of Technology and Sciences, Madanapalle, raja.narayana@gmail.com

B Sasikala
Madanapalle Institute of Technology and Sciences, Madanapalle, b.sasikala@gmail.com

N Balakrishna
Madanapalle Institute of Technology and Sciences, Madanapalle, n.balakrishna@gmail.com

Efficient Multi Dimensional Nodal Fuzzy Search for Information Management System

M Raja Narayana, B Sasikala & N Balakrishna

Madanapalle Institute of Technology and Sciences, Madanapalle

Abstract – We propose a novel multi-dimensional search approach that allows users to perform fuzzy searches for structure and metadata conditions in addition to keyword conditions. Our techniques individually score each dimension and integrate the three dimension scores into a meaningful unified score. We also design indexes and algorithms to efficiently identify the most relevant files that match multi-dimensional queries. We perform a thorough experimental evaluation of our approach and show that our relaxation and scoring framework for fuzzy query conditions in non content dimensions can significantly improve ranking accuracy. We also show that our query processing strategies perform and scale well, making our fuzzy search approach practical for every day usage.

I. INTRODUCTION

The amount of data stored in personal information management systems is rapidly increasing, following the relentless growth in capacity and dropping price of storage. This explosion of information is driving a critical need for powerful search tools to access often very heterogeneous data in a simple and efficient manner. Such tools should provide both high-quality scoring mechanisms and efficient query processing capabilities. Numerous search tools have been developed to perform keyword searches and locate personal information stored in file systems, such as the commercial tools Google Desktop Search and Spotlight. However, these tools usually support some form of ranking for the textual part of the query—similar to what has been done in the Information Retrieval (IR) community—but only consider structure (e.g., file directory) and metadata (e.g., date, file type) as filtering conditions. THE amount of data stored in personal information management systems is rapidly increasing, following the relentless growth in capacity and dropping price of storage.

II. UNIFIED MULTI-DIMENSIONAL SCORING

In this section, we present our unified framework for assigning scores to files based on how closely they match query conditions within different query dimensions. We distinguish three scoring dimensions: *content* for conditions on the textual content of the files, *metadata* for conditions on the system information related to the files, and *structure* for conditions on the directory path to access the file. We represent files and their associated metadata and structure information as XML documents. We use a simplified version of XQuery to express metadata and structure conditions in addition to keyword-based content conditions.

2.1 Scoring Content

We use standard IR relaxation and scoring techniques for content query conditions [30]. Specifically, we adopt the *TF·IDF* scoring formulas from Lucene [6], a state-of-theart keyword search tool. These formulas are as follows:

$$Score(Q,f) = \sum_{t \in Q} \frac{scorec.tf(t,f) \times scorec.idf(t)}{NormLength(f)}$$

$$Scorec,tf(t,f) = \sqrt{\text{No}}\ \text{times t occurs in}\ f$$

$$scorec,idf(t) = 1 + \log\left(\frac{N}{1+Nt}\right)$$

Where *Q* is the content query condition, *f* is the file being scored, *N* is the total number of files, *Nt* 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.

2.2 Scoring Metadata

We introduce a hierarchical relaxation approach for each type of searchable metadata to support scoring. For example, Figure 1 shows (a portion of) the relaxation levels for file types, represented as a DAG3. Each leaf represents a specific file type (e.g., pdf files). Each internal node represents a more general file type that is the union of the types of its children (e.g., *Media* is the union of *Video*, *Image,* and *Music*) and thus is a relaxation of its descendants. A key characteristic of this hierarchical representation is *containment*; that is, the set of files matching a node must be equal to or subsume the set of files matching each of its children nodes. This ensures that the score of a 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 relaxed form (see Equation 4 below). We then say that a metadata condition *matches* a DAG node if the node's range of metadata values is equal to or subsumes the query condition. For example, a file type query condition specifying a file of type "*.cpp" would match the nodes representing files of type "Code", files of type "Document", etc. A query condition on the creation date of a file would match different levels of time granularity, e.g., day, week or month. The nodes on the path from the deepest (most restrictive) matching node to the root of the DAG then represent all of the relaxations that we can score for that query condition. Similarly, each file *matches* all nodes in the DAG that is equal to or subsumes the file's metadata value.

2.3 Scoring Structure

Most users use a hierarchical directory structure to organize their files. When searching for a particular file, a user may often remember some of the components of the containing directory path and their approximate ordering rather than the exact path itself. Thus, allowing for some approximation on structure query conditions is desirable because it allows users to leverage their partial memory to help the search engine locate the desired file. Our structure scoring strategy extends prior work on XML structural query relaxations [4], [5]. Specifically, the node inversion relaxation introduced below 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:

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

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

• **Node Inversion** 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 except for the *root* and ∗ nodes. A permutation combines adjacent nodes, or node groups, into a single node group while preserving the relative order of edges in *P*. For example, applying node inversion on *b* and *c* from */a/b/c* would result in */a/*(*b/c*), allowing for both the original query condition as well as */a/c/b*. The (*b/c*) part of the relaxed condition */a/*(*b/c*) is called a *node group*.

• **Node Deletion** is used to drop a node from a path. Node deletion can be applied to any path query or node group but cannot be used to delete the *root* node or the ∗ node. To delete a node *n* in a path query *P*: – If *n* is a leaf node, *n* is dropped from *P* and $P - n$ is extended with *//*∗. This is to ensure containment of the exact answers to *P* in the set of answers to *P* _, and monotonicity of scores.– If *n* is an internal node, *n* is dropped from *P* and *parent*(*n*) and *child*(*n*) are connected in *P* with *//*.For example, deleting node *c* from *a/b/c* results in *a/b//*∗ because *a/b//*∗ is the most specific relaxed path query containing *a/b/c* that does not contain *c*. Similarly, deleting *c* from *a/c/b//*∗ results in *a//b//*∗. To delete a node *n* that is within a node group *N*

2.4 Score Aggregation

We aggregate the above single-dimensional scores into a unified multi-dimensional score to provide a unified ranking of files relevant to a multi-dimensional query. To do this, we construct a query vector, *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 aggregated score of file *F*. In its current form, this is simply a linear combination of the component scores with equal weighting. The vector projection method, however, provides a framework for future exploration of more complex aggregations.

III. QUERY PROCESSING

We adapt an existing algorithm called the Threshold Algorithm(TA) [15] to drive query processing. TA uses a threshold condition to avoid evaluating all possible matches to a query, focusing instead on identifying the *k* best answers. It takes as input several sorted lists, each containing the system's objects (files in our scenario) sorted in descending order according to their relevance scores for a particular attribute (dimension in our scenario), and dynamically accesses the sorted lists until the threshold condition is met to find the *k* best answers. Critically, TA relies on *sorted* and *random accesses* to retrieve individual attribute scores. Sorted accesses, that is, accesses to the sorted lists mentioned above, require the files to be returned in descending order of their scores for a particular dimension. Random accesses require the computation of a score for a particular dimension for any given file. Random accesses occur when TA chooses a file from a particular list corresponding to some dimension, then needs the scores for the file in all the other dimensions to compute its unified score.

3.1 Evaluating Content Scores

As mentioned in Section 2.1, we use existing *TF·IDF* methods to score the content dimension. Random accesses are supported via standard inverted list implementations, where, for each query term, we can easily look up the term frequency in the entire file system as well as in a particular file. We support sorted accesses by keeping the inverted lists in sorted order; that is, for the set of files that contain a particular term, we keep the files in sorted order according to their TF scores, normalized by file size, for that term.4 We then use the TA algorithm recursively to return files in sorted order according to their content scores for queries that contain more than one term.

3.2 Evaluating Metadata Scores

Sorted access for a metadata condition is implemented using the appropriate relaxation DAG index. First, exact matches are identified by identifying the deepest DAG node *N* that matches the given metadata condition (see Section 2.2). Once all exact matches have been retrieved from *N*'s leaf descendants, approximate matches are produced by traversing up the DAG to consider more approximate matches. Each parent contains a larger range of values than its children, which ensures that the matches are returned in decreasing order of metadata scores. Similar to the content dimension, we use the TA algorithm recursively to return files in sorted order for queries that contain multiple metadata conditions. Random accesses for a metadata condition require locating in the appropriate DAG index the closest common ancestor of the deepest node that matches the condition and the deepest node that matches the file's metadata attribute. This is implemented as an efficient DAG traversal algorithm.

IV. 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:

- The DAGs representing relaxations of structure conditions [4], [24] 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.
- The *TA* algorithm requires efficient sorted and random access to the single-dimension scores (Section 3). In particular, random accesses can be very expensive. We need efficient indexes and traversal algorithms that support both types of access.

4.1 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 DAG structure. Scoring an entire query relaxation DAG can be expensive as they grow exponentially with the size of the query condition. For example, there are 5, 21, 94, 427, and 1946 nodes in the respective complete DAGs for query conditions */a*, */a/b*, */a/b/c*, */a/b/c/d*, */a/b/c/d/e*. However, in many cases, enough query matches will be found near the top of the DAG, and a large portion of the DAG will not 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.

4.2 Improving Sorted Accesses:

Evaluating queries with structure conditions using the lazy DAG building algorithm can lead to significant query evaluation times as it is common for multi-dimensional top*k* processing to access very relaxed structure matches, i.e., matches to relaxed query paths that lay at the bottom of the DAG, to compute the top-*k* answers. An interesting observation is that not every possible relaxation leads to the discovery of new matches. This is formalized in Theorem 1 *Theorem 1:* Given the structural *scoreidf* function defined in Equation 6, if a query path *P* _ is a relaxed version of another query path *P*, and *scoreidf* (*P*_) = *scoreidf* (*P*) in the structure DAG, any node *P* __ on any path from *P* to *P*_ has the same structure score as *scoreidf* (*P*), and *F*(*P*_) = *F*(*P*__) = *F*(*P*), where *F*(*P*) is the set of files matching query path *P*. *Proof:* (Sketch) If *scoreidf* (*P*_) = *scoreidf* (*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* _) ⊇ *F*(*P*__) ⊇ *F*(*P*) and *NP*_ ≥ *NP*__ ≥ *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, Dynamic *DAG*, based on the above idea. It includes two steps: (a) starting at a node corresponding to a query path *P*, the algorithm performs a depth-first traversal and scoring of the DAG until it finds a parentchild pair, *P* _ and *child*(*P*_), where *scoreidf* (*child*(*P*_)) < *scoreidf* (*P*); and (b) score each node *P* __ at the same

Algorithm 1 Dynamic DAG(srcNode)

```
1.  $s$ ⇐ getScore(srcNode)
2.  currentNode ⇐ srcNode
3.  **loop**
4.  targetDepth ⇐ getDepth(currentNode)
5.  childNode ⇐ firstChild(currentNode)
6.  **if** getScore(childNode) _= $s$ or
    hasNoChildNodes(childNode) **then**
7.  exit loop
8.  currentNode ⇐ childNode
9.  **for** each $n$ s.t. getDepth($n$) = targetDepth and
    getScore($n$) = $s$ **do**
10. Evaluate bottom-up from $n$ and identify ancestor
    node set
    $S$ s.t. getScore($m$) = $s$, $\forall m \in S$
11. **for** each $m \in S$ **do**
12. **for** each $n$_ on path $p \in$ getPaths($n,m$) **do**
13. setScore($n$_, $s$)
14. setSkippable($n$_)
15. **if** notSkippable($m$) **then**
16. setSkippable($m$)
```

V. CONCLUSIONS

We presented a scoring framework for multi-dimensional queries over personal information management systems. Specifically, we defined structure and metadata relaxations and proposed *IDF*-based scoring approaches for content, metadata, and structure query conditions. This uniformity of scoring allows individual dimension scores to be easily aggregated.

REFERENCES

[1] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03), 2003.

[2] 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.

[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.

[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] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR technologies: What is the

sound of one hand clapping? In Proc. Of the Conference on Innovative Data Systems Research (CIDR), 2005.

[13] J. Chen, H. Guo, W. Wu, and C. Xie. Search Your Memory! – An Associative Memory Based Desktop Search System. In Proc. of the ACM Intl. Conference on Management of Data (SIGMOD), 2009.

[14] 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.

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

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

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

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

[19] K. A. Gyllstrom, C. Soules, and A. Veitch. Confluence: Enhancing Contextual Desktop Search. In Proc. of the ACM Intl. Conference on Research and Development in Information Retrieval (SIGIR), 2007.

◈◈◈