Relaxation in Text Search using Taxonomies

Marcus Fontoura Vanja Josifovski Ravi Kumar Christopher Olston
Andrew Tomkins Sergei Vassilvitskii
Yahoo! Research
701 First Ave.
Sunnyvale, CA 94089.
{marcusf, vanjaj, ravikumar, olston, atomkins, sergei}@yahoo-inc.com

ABSTRACT

In this paper we propose a novel document retrieval model in which text queries are augmented with multi-dimensional taxonomy restrictions. These restrictions may be relaxed at a cost to result quality. This new model may be applicable in many arenas, including multifaceted, product, and local search, where documents are augmented with hierarchical metadata such as topic or location. We present efficient algorithms for indexing and query processing in this new retrieval model. We decompose query processing into two sub-problems: first, an online search problem to determine the correct overall level of relaxation cost that must be incurred to generate the top k results; and second, a budgeted relaxation search problem in which all results at a particular relaxation cost must be produced at minimal cost. We show the latter problem is solvable exactly in two hierarchical dimensions, is NP-hard in three or more dimensions, but admits efficient approximation algorithms with provable guarantees. We present experimental results evaluating our algorithms on both synthetic and real data, showing order of magnitude improvements over the baseline algorithm.

1. INTRODUCTION

Information retrieval (IR) systems have developed specialized data structures and algorithms to perform a specific task: ranked retrieval of documents. These systems are increasingly being called upon to incorporate more complex processing into query evaluation. Some extensions, such as query expansion, can be handled cleanly in the existing model. Others, such as static scoring, may be incorporated with only small changes to the underlying system. But an increasingly prominent set of desired extensions do not naturally fit within the traditional document model. Typical examples are *local search*, in which the user is interested only in geographically proximate results [9], *multifaceted product search*, in which product metadata is effective to restrict search along many dimensions [35], and *social search*, in which an endorsement by another user may alter the ranking of an object based on the relationship between the user and the endorser.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '08, August 24-30, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

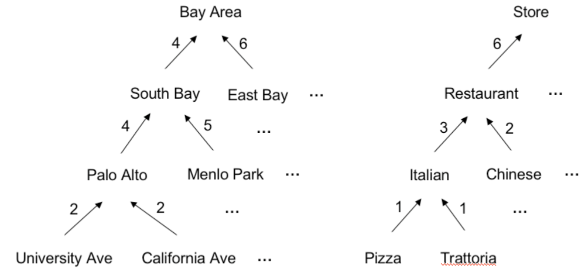

Figure 1: Hierarchical document taxonomies.

Let us consider a motivating example. Sue is in University Ave., Palo Alto, California. She feels the sudden urge to indulge in deep dish pizza. She enters an internet cafe, navigates to a purveyor of local search, and queries for such a restaurant in her vicinity. The local search engine may take into account the following factors.
(1) Sue has indicated that she wishes the web page of matching institutions to contain the term "deep dish."
(2) Objects known to the local search engine are labeled with a category, such as restaurant, or more specifically pizza parlor.
(3) Sue is known to be on University Ave., which is part of Palo Alto, which is part of the South Bay.

The scenario contains two types of information. The first, in item (1), is a traditional textual query that is amenable to existing techniques. The second, in items (2) and (3), is a set of desired characteristics of the result, expressed as leaves of a tree such as the ones illustrated in Figure 1. The result should be a restaurant of type pizza parlor, but in a pinch, perhaps an Italian restaurant will do. The location should be University Ave., but again, perhaps a restaurant located in nearby Menlo Park would be appropriate if it is a good match to the other constraints. Thus, each of these restrictions exists within a hierarchical structure allowing relaxation at some cost.

1.1 Problem definition

Multi-dimensional relaxation may be phrased in many ways: one may relax a location by specifying a ball of growing radius around that location, or may relax a phrase using linguistic measures of phrase similarity. In our formulation, we focus specifically on relaxation via multiple hierarchies, for three reasons. First, hierarchies capture many important notions of relaxation that are used today. Second, other forms of relaxations may be cast as hierarchies with some loss (see, for instance, [4]). In product search, for instance,

the price values are normally mapped into ranges, which can easily be represented as taxonomies. And finally, the restriction allows us to phrase a clean combinatorial problem that is amenable to algorithmic techniques.

We define a *taxonomy* as a tree whose edges have nonnegative weights. For a taxonomy T, each document d in the corpus is associated with exactly one node of T, denoted $\text{topic}(d) \in T$[1]. Let $T_1, \ldots, T_m$ be the taxonomies and let $\text{topic}_j(d) \in T_j$ denote the node in the j-th taxonomy associated with document d.

Figure 1 shows two taxonomies for our running example. The weights determine the cost of relaxation in an additive fashion. For a query seeking a pizza restaurant on University Ave., a non-pizza Italian restaurant located in the South Bay but not in Palo Alto would incur a total relaxation cost of $1+2+4 = 7$. We assume these weights have been assigned by a domain expert—our algorithms allow any nonnegative weights and allow specification of the weights at runtime, so it is possible to evaluate queries with user-specific relaxation costs. We also assume that these weights have been normalized across taxonomies in a way that their addition yields the correct query semantics.

User-entered queries have two components: (1) a text component and (2) a set of taxonomy nodes. More formally, a query Q consists of text keywords $\text{keyw}(Q)$ and a vector of taxonomy nodes $\text{topic}(Q) = \langle \text{topic}_1(Q), \ldots, \text{topic}_m(Q) \rangle$, where $\text{topic}_j(Q) \in T_j$. The answer to a query consists of the top k results, ranked in increasing order according to the following scoring function:

$$\begin{aligned} \text{score}(d, Q) &= \alpha\, \text{static}(d) + \beta\, \text{text}(d, \text{keyw}(Q)) \\ &\quad + \gamma\, \text{tax}(d, \text{topic}(Q)). \end{aligned} \tag{1}$$

Here,

$\text{static}(d)$ returns the query-independent "importance" score for document d (e.g., d's PageRank [31]);

$\text{text}(d, \text{keyw}(Q))$ returns the text-based relevance score for document d with respect to keywords $\text{keyw}(Q)$; and

$\text{tax}(d, \text{topic}(Q))$ returns a taxonomy score, i.e., a collective *relaxation cost* for document d with respect to a list of taxonomy nodes $\text{topic}(Q)$.

(For all score components, lower is better. The variables α, β, and γ are weights for the different score components.) We set

$$\text{tax}(d, \text{topic}(Q)) = \sum_{j=1}^{m} \text{tax}_j(\text{topic}_j(d), \text{topic}_j(Q)),$$

i.e., the sum over relaxation costs in each taxonomy, where $\text{topic}_j(Q)$ denotes the query node in taxonomy T_j. Relaxation cost in the j-th taxonomy is defined as

$$\text{tax}_j(t_d, t_Q) = \text{wdist}_j\left(t_Q, \text{lca}(t_Q, t_d)\right),$$

where $t_Q = \text{topic}_j(Q)$, $t_d = \text{topic}_j(d)$, $\text{lca}(\cdot, \cdot)$ is the least-common ancestor function, and $\text{wdist}_j(\cdot, \cdot)$ gives the sum of the edge weights along the path between two taxonomy nodes in T_j.[2] The above equation should be read as follows: if the query specifies University Ave. and the document is located at University Ave., the relaxation cost is 0. But if the document is located on California Ave., also in Palo Alto, then the lca is Palo Alto and the relaxation cost depends on the distance in the tree from University Ave. to Palo Alto, and so forth.

There are three main subproblems that need to be addressed in order to fully support the new document retrieval model we propose: (1) creation of taxonomies and selection of appropriate taxonomy weights, (2) mapping of query terms and user information into taxonomy nodes, and (3) efficient indexing and query processing. In this paper we focus on the third subproblem: our goal is to index taxonomies with their weights and efficiently find top k answers for query Q based on our scoring function $\text{score}(d, Q)$.

1.2 Our approach

A natural IR-based strategy to address indexing and query processing would be to use a text index to obtain establishments relevant to "deep dish pizza," and then post-process using the metadata, i.e., restaurant classification data, geographical data, and user preferences. However, text matching may not represent the most selective access path (especially if relaxed text matching semantics are employed). Given that queries have multiple access paths to data, it makes sense to consider a broader space of evaluation strategies, in the spirit of database query optimization. However, care must be taken in our domain: the presence of multiple potential relaxations results in a planning space that is exponential in the number of taxonomies.

Our approach is to extend the text index to also include the taxonomy nodes, and process the text and taxonomy portions of the query simultaneously via the index. In particular, we create a posting list[3] for each taxonomy node. To process a query we select initial taxonomy nodes and begin to traverse their posting lists. As results begin to emerge, we adjust the degree of relaxation (i.e., move up or down in the taxonomy), with the goal of scanning the shortest posting lists possible (i.e., posting lists for low taxonomy nodes) while still producing the top k results. In the case of multiple taxonomies we may choose to issue multiple simultaneous relaxation strategies (i.e., moving up in taxonomy one versus moving up in taxonomy two), so as to avoid overly relaxed strategies that produce excessive matches.

1.3 Main contributions

(1) The formalization of a novel document retrieval model, in which textual queries are augmented with multi-dimensional taxonomy restrictions. These restrictions may be relaxed at a cost to result quality.

(2) Given taxonomies along with weights, algorithms for indexing (Section 4) and query processing (Section 5).

(3) A formal treatment of the problem of query processing at a fixed relaxation. We show that this problem problem is solvable exactly in two dimensions, is NP-hard in three or more dimensions, but admits efficient approximation algorithms with provable guarantees (Section 5.3).

(4) Experimental results evaluating our algorithms on both synthetic and real data, showing order of magnitude improvements over the baseline algorithm (Section 6).

[1] For simplicity, we omit the cases in which a taxonomy does not contain all documents or a document is associated with multiple nodes of a given taxonomy, but our approach can be easily extended to handle these cases.

[2] This apparently asymmetric measure of relaxation cost may be converted to a symmetric version which returns the same ordering of results in a straightforward manner.

[3] Section 3 provides background information on posting lists and inverted text indexes.

2. RELATED WORK

Query relaxation has been studied in the context of XML and semi-structured databases, e.g., [36]. The focus of this work is how to relax query constraints in order to deal with imprecise document models and heterogeneous schemas. It does not consider relaxation via taxonomies, which is our concern.

Document taxonomies are already being used in some commercial IR engines. Local search in today's web search engines uses a location taxonomy to find out the relevant documents in close proximity to a given location. While there is no published work from the major search engines describing their current implementations of local search, we assume that these do not generalize to multiple taxonomies as described in this paper. Several indexing and query processing algorithms for local search have been proposed in [9]. In that work, queries have two components—a textual component and a geographic component. The algorithms proposed in [9] are based on space-filling curves and do not use taxonomies. We view that work as complementary to ours, since the taxonomy component of our scoring function can be easily integrated into their processing model.

Another example of use of taxonomies on the web is product search where products are classified into taxonomies to aid the browsing and search; see [35] for an overview. Multifaceted search products such as Endeca (`endeca.com`), i411 (`i411.com`), FacetMap (`facetmap.com`), Flamenco (`flamenco.berkeley.edu`), aim to improve the search and navigation of document collections by allowing the user to drill down in the set of documents returned by a query.

There has been much recent work on keyword search over structured or semi-structured databases, e.g., [2, 16, 24, 26, 34]. Typically the aim is to find database fragments (i.e., XML subtrees or joined relational tuples) such that each fragment contains all query keywords and is minimal in some respect (e.g., minimal in size). In our work we focus on how to leverage metadata about documents, which in our case occurs in the form of one or more independent taxonomies. A query does not just consist of keywords, but also specifies desired taxonomy positions.

A taxonomy of attributes has been used in OLAP systems for aggregation and exact queries [21, 25]. In this setting, it has been mostly used to determine the right granularity of aggregation. However, increasingly sophisticated algorithmic approaches and problem formulations have been applied to the same underlying data model [1, 12, 13, 14]. In the OLAP context, systems have been proposed to consider automated extraction of hierarchical metadata from documents [11, 27]. Automated extraction also raises novel problems for query processing over such uncertain data [8].

The query processing algorithms presented in this paper are similar in spirit to the WAND adaptive algorithm [6] and Threshold algorithm in [15]. Like WAND, our algorithms start assuming no knowledge about the distribution of the results and improve their performance as data is seen during query processing. The presence of text and static scores in our case makes the algorithms in [15] not applicable. For efficiently processing text tokens in the document-at-a-time query evaluation model, described in Section 3, all the lists in the text index must be in the same order. The algorithms proposed in [15] take as input several lists, each of them sorted by different criteria.

3. BACKGROUND

In this section we briefly review some basic IR concepts and terminology.

Inverted index. Most IR systems use inverted indexes as their main data structure for full-text indexing [33]. There is a considerable body of literature on efficient ways to build inverted indexes [3, 5, 18, 23, 30, 33] and evaluate full-text queries using them [6, 29, 33].

In this paper we assume an inverted index structure. The occurrence of a term t within a document d is called a *posting*. The set of postings associated to a term t is stored in a *posting list*. A posting has the form $\langle$*docid, payload*$\rangle$, where *docid* is the document ID of d and where the payload is used to store arbitrary information about each occurrence of t within d.

Each posting list is sorted in increasing order of *docid*. Often, B-trees [20] or skip lists are used to index the posting lists [18, 30]. This facilitates searching for a particular *docid* within a posting list, or for the smallest *docid* in the list greater than a given *docid*.

Free-text queries. Most IR systems support free-text queries, allowing Boolean expressions on keywords and phrase searches. In this paper we assume the document-at-a-time query evaluation model (DAAT) [32], commonly used in web search engines. In DAAT, the documents that satisfy the query are usually obtained via a zig-zag join [20] of the posting lists of the query terms. To evaluate a free-text query using a zig-zag join, a *cursor* C_t is created for each term t in the query, and is used to access t's posting list. $C_t.docid$ and $C_t.payload$ access the *docid* and *payload* of the posting on which C_t is currently positioned. During a zig-zag join, the cursors are moved in a coordinated way to find the documents that satisfy the query. Two basic methods on a cursor C_t are required to do this efficiently:

(1) C_t.getNext() advances C_t to the next posting in its posting list.

(2) C_t.fwdBeyond(d) advances C_t to the first posting in its posting list whose *docid* is greater than or equal to d. Since posting lists are ordered by *docid*, this operation can be done efficiently.

Scoring. Once a zig-zag join has positioned the cursors on a document that satisfies the query, the document is scored. The final score for a document usually contains a query-dependent textual component, which is based on the document similarity to the query, and a query-independent static component, which is based on the *static rank* of the document. In most IR systems the textual component of the score follows an additive scoring model like *tf* $\times$ *idf* for each term, whereas the static component can be based on the connectivity of web pages, as in PageRank [31], or on other factors such as source, length, creation date, etc. In this paper the score also has a third component, which is the taxonomy score.

4. INDEX STRUCTURE TO SUPPORT EFFICIENT RETRIEVAL

In this paper we extend the use of the inverted index to allow queries for any node t_j of any taxonomy T_j; this is in addition to the usual text queries. To support the taxonomy node queries in an efficient manner, we choose to add one additional posting list per taxonomy node. Each of these lists contains one posting for each document that

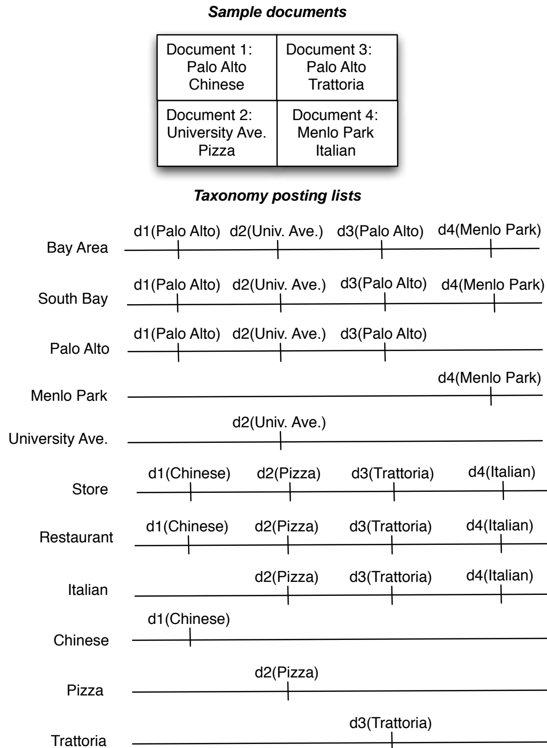

Figure 2: Sample documents with their taxonomy values and corresponding taxonomy posting lists. All the posting lists are sorted by *docids*.

belongs to the corresponding subtree of that taxonomy node (i.e., $d \in T_j|_{t_j}$), no matter what their location in other taxonomies. In order to allow for precise ranking during query evaluation, the payloads of postings in these special posting lists identifies the exact placement of the document d in the taxonomy tree T_j. The postings in these postings lists are sorted by *docid*—in the same order used for the text posting lists. Therefore these lists can be used in Boolean queries in the same manner as the text posting lists. The fact that all posting lists are ordered by *docid* is crucial for the efficient implementation of DAAT query processing algorithms. Figure 2 shows the taxonomy posting lists for four sample documents. The payload values are shown between parenthesis. For clarity we show the name of the taxonomy node as the payload value, but for efficiency these values should be encoded in the inverted index, for instance, using numerical ids. The Italian posting list shown in Figure 2, for instance, is a combination of the Pizza and Trattoria posting lists plus the documents that appear on the Italian node itself (Document 4).

Retrieval is accomplished in the usual way for DAAT query processing algorithms, by opening a cursor into each relevant posting list, and then advancing the cursors in a coordinated fashion to find documents that occur in the intersection of the lists. For example, to find restaurants in Palo Alto, we perform a Boolean search for "Palo Alto AND Restaurants." When a document is returned as the result of such a query, we know its position in all the taxonomies by looking at its payload values, and we can therefore compute its relaxation cost.

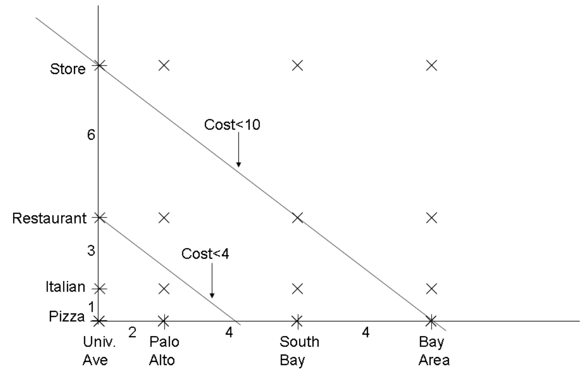

Figure 3: Query evaluation options.

The overhead required by this indexing scheme is minimal. First, the number of terms in the index increases by the number of nodes in the union of all taxonomies; this number may range into the tens of thousands in extreme cases, compared to the millions of words that are common in large document indexes. Second, we must add postings entries for each document in each tree. However, the document will occur in a number of nodes corresponding to the sum of the average depths of the trees: this will typically be measured in tens, rather than the many hundreds or thousands of words present in an average web document. Thus, at least in the context of web search, the expected overhead of such a scheme is below 1%; see [22] for a system implementing this scheme at minimal cost in another context.

5. QUERY PROCESSING

The overall objective of the query engine is to find the k documents of least cost under our scoring model (Section 1.1), while minimizing the total retrieval cost incurred. For ease of exposition, we will focus on the case in which there is no text query and no static score; that is, the cost of a result is exactly the relaxation cost. The text and static score components may be incorporated easily.

We first describe the space of possible relaxation costs (Section 5.1). We then describe the query processing step, decomposing the problem into two sub-problems: first, an online search problem to determine the overall level of relaxation cost (Section 5.2); and second, a budgeted relaxation search problem in which all results at a particular relaxation cost must be produced at minimal cost (Section 5.3).

5.1 The space of relaxation options

Let us consider the query "University Ave. and Pizza" of our running example. We show the space of possible two-dimensional relaxations in Figure 3. The bottom-left corner represents the query point q. The possible relaxations in taxonomy T_1 are placed on the x-axis and the possible relaxations in T_2 are placed on the y-axis. For each axis, tick marks indicate the points for which a relaxation exists. For each node $\text{topic}_j(Q)$, the define the weighted path in T_j from $\text{topic}_j(Q)$ to the root as the *relaxation path* of Q in T_j. The symbol '$\times$' marks indicate possible relaxations in the cartesian product of the two relaxation paths, i.e., points for which both coordinates lie at a tick mark. The relaxation cost of a point at position (x, y) is $x + y$. For instance, the top right '$\times$' mark corresponds to the node (Bay Area, Store) and its relaxation cost is $10 + 10 = 20$.

In general, with $m \geq 0$ taxonomies we have an m-dimensional grid, where each grid point $(t_1, \ldots, t_m)$ is such that $t_j \in T_j$. Each grid point is therefore an element of $T_1 \times \cdots \times T_m$, and each t_j is on the path from $\text{topic}_j(Q)$ to the root of T_j. The relaxation cost of a point is exactly its L_1 norm, the sum of its coordinates.

Consider a grid point $(t_1, \ldots, t_m)$. This point implicitly corresponds to a subset of documents given by the intersection of the taxonomy nodes at each point; for example, all objects that have both geography Palo Alto and restaurant type Italian. Let $T|_t$ denote the subtree of T rooted at node t, and let $\text{docs}(t)$ denote the set of documents whose topic in T lies in the subtree $T|_t$. For a grid point $(t_1, \ldots, t_m)$, we define:

$$\text{docs}(t_1, \ldots, t_m) = \cap_{j=1}^{m} \text{docs}(t_j).$$

We will require one additional piece of notation. As the relaxation cost of a point is the sum of its coordinates, we observe that lines of slope -1 represent thresholds of relaxation cost: all points below the line have relaxation cost at most the x-intercept of the line. Figure 3 shows lines with relaxation cost 4 and 10. In general, the region of relaxation cost at most some budget b will be a simplex $S(b)$; in two dimensions, the simplex is a triangular region defined by the two axes and the appropriate diagonal line of slope -1. Adjusting the slope is equivalent to scaling the weights of the two taxonomies; this effect can alternatively be accomplished by modifying the taxonomy edge weights, effectively scaling one of the axes, so without loss of generality we only consider lines of slope -1. Formally,

$$S(b) = \{(t_1, \ldots, t_m) \mid b \geq \sum_{j=1}^{m} \text{tax}_j(t_j, \text{topic}_j(Q))\},$$

i.e., $S(b)$ contains all points in the grid that have relaxation cost at most b. We use the natural notation

$$\text{docs}(S(b)) = \cup_{(t_1,\ldots,t_m) \in S(b)} \text{docs}(t_1, \ldots, t_m),$$

to denote the set of all documents present in the nodes defined by the simplex $S(b)$; this represents all documents with relaxation cost at most b.

As explained in Sections 5.2 and 5.3, evaluation of a user's top-k query will be performed by issuing a sequence of retrieval commands to an underlying index structure. The main retrieval primitive we employ returns $\text{docs}(t_1, \ldots, t_m)$ for a given point $(t_1, \ldots, t_m)$.

5.2 Top-k relaxation search

In this section we study how best to search over the space of relaxation budgets, in order to retrieve the top k documents as cheaply as possible. Recalling that $S(b)$ represents the simplex of points with relaxation cost $\leq b$, and that $\text{docs}(S(b))$ represents the set of documents with at most this relaxation cost, we can state our formal goal as follows: find the minimal relaxation budget b^* such that $|\text{docs}(S(b^*))| \geq k$. We will define a series of "levels" that are the natural candidate relaxation budgets to consider. In terms of our visualization of the problem given in Figure 3, imagine sliding a line of slope -1 from the origin up and to the right. Each time the line intersects a tick mark, there is a candidate budget corresponding to the x-intercept that may contain more documents than any smaller budget. In Figure 3, such lines are shown corresponding to budgets 4 and 10; of course, other candidate budgets exist, but are not drawn in. Formally, we will search a precomputed list of levels $B = b_1, \ldots, b_L$, with $b_j \in [0, \sum_{j=1}^{m} \text{tax}_j(\text{root}(T_j), \text{topic}_j(Q))]$ such that for any $b' \in [b_j, b_{j+1})$ we have $\text{docs}(S(b')) = \text{docs}(S(b_j))$. Levels in B occur in increasing order of budget and L is the maximum level.

Depending on the nature of the taxonomy weights, this set of levels may be unnecessarily large, so it is possible to prune the set of levels we consider to a more manageable subset. In our experiments, we will restrict our attention to levels in which the line intersects an axis tick mark, and declare these to be the candidate relaxation budgets. Our goal is then to search through these budgets for the smallest one that yields k documents.

In the next subsection we present a family of algorithms based on a *conservative* search method that moves to a new level once it is 100% certain that the current level is unsuitable. It is possible to use an *aggressive* search method that may choose to abandon a level early based on statistical evidence; we leave it as an interesting future direction to investigate.

5.2.1 Conservative search methods

Let us define the main data structures for our algorithms. Let ℓ denote the current level and let R denote the current set of results; we store the results in a heap so that it is becomes efficient to keep the top k results.

Recall that at each level ℓ, the documents at that level are given by $\text{docs}(S(b_\ell))$. We begin processing documents in this set by traversing the underlying index using one or more retrieval queries as dictated by the method for budgeted relaxation search (described in Section 5.3) with budget b_ℓ.

If we finish processing all documents at level ℓ, but have not obtained k results, then we need to consider a more relaxed level (larger value of ℓ). In our running example, for query (University Ave., Pizza), suppose $k = 2$ and we begin with $\ell = 0$. We find exactly one document whose relaxation cost is 0 (Document 2). We are trying to find at least two documents (since $k = 2$), so we must try again with a larger value of ℓ, say $\ell = 1$.

In general, larger values of ℓ give us more results, and hence a greater chance of getting at least k results—but at a greater query cost. The goal is to perform an iterative search over different ℓ values, to converge on one that gives us k results without incurring excessive cost.

We have a generic procedure for performing this search for the best level. Our generic search procedure is governed by two functions. The first function, initialLevel(), controls the level at which we begin our search. The second function, getNextLevel(ℓ), picks a new level to explore, if we decide that the current level ℓ is not a good choice. Different manifestations of these functions yield different search strategies.

We first give the generic search algorithm, which is parameterized by the functions initialLevel() and getNextLevel(), and then describe specific algorithms that use specific manifestations of these functions.

Generic search procedure. We now present a generic algorithm that performs processing given the two functions initialLevel() and getNextLevel(). The pseudocode is given next.

Algorithm ConservativeSearch (Q)
1. ℓ = initialLevel()
2. levelDone = false
3. while ($|R| < k \vee \neg$ levelDone)
4. levelDone = processNextDoc (Q, R, b_ℓ)
5. if (($|R| \geq k$)$\vee$ levelDone)
6. ℓ = getNextLevel (ℓ)

The basic idea behind this algorithms is as follows. Once the level is chosen, in line 4 the algorithm invokes the function processNextDoc(Q, R, b_ℓ) to retrieve documents for that level, i.e., documents of cost $\leq b_\ell$. It returns a Boolean value (levelDone) indicating if there are more documents to be processed for that level.

As the level changes, processNextDoc() must restart this enumeration as necessary to process documents in order from the requested level. If during processing of the next document the set R of results reaches size k and all the documents in R have total cost at most b_ℓ this is an indication that specialization is possible without compromising the desired number of results. In this case, the level will be specialized (ℓ is decreased) in the call to getNextLevel() (line 6—we enter the if since $|R| \geq k$). If processNextDoc() finishes scanning all the documents in docs$(S(b_\ell))$ and still $|R| < k$ the level needs to be relaxed (ℓ is increased), which will happen in the call to getNextLevel() (line 6—we enter the if since levelDone is true).

We now describe three natural instantiations of this template, which are realized by different variants of the functions initialLevel() and getNextLevel(). It is important to note that in all these instantiations, the top k scoring documents are returned.

(1) Bottom-up search. We start with the most specific query possible, such as (University Ave., Pizza). If there are at least k documents in the current level ℓ, we are done. Otherwise, we relax by increasing the current level, e.g., to $\ell + 1$. In this case, we need to restart the querying process, issuing a new query to the budgeted relaxation search algorithm with budget $b_{\ell+1}$. We only call getNextLevel() (line 6) to relax the query if levelDone is true and we still did not find k results.

Let us consider again the documents from Figure 2. The bottom-up search for the top-2 documents is illustrated in Figure 4. For query (University Ave., Pizza) Document 1 has relaxation cost $2+4 = 6$, Document 2 has relaxation cost 0, Document 3 has relaxation cost $2+1 = 3$, and Document 4 has cost $6 + 1 = 7$. Therefore, if we are interested in the top-2 documents we want to retrieve Documents 2 and 3, in this order.

As shown in Figure 4 we start processing at $\ell = 0$ with posting lists University Ave. and Pizza and retrieve Document 2 from the zig-zag join of these two posting lists. Since we are interested in top-2 results, we have $k = 2$ and we get to line 5 in the algorithm with $|R| = 1$ and levelDone = true. At this point we need to relax the query in order to try to obtain the second result we are looking for. We then set $\ell = 1$ and use posting lists University Ave. and Italian. This join generates no results different than Document 2, since the only document in the University Ave. posting list is Document 2, and therefore we need to generalize again. The next level to try is $\ell = 3$, using posting lists Palo Alto and Italian (there is no posting list combination that we can try for $\ell = 2$ in this example). The zig-zag join using these

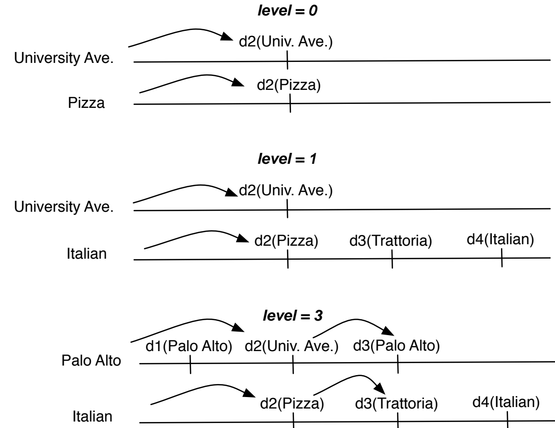

Figure 4: Bottom-up search example.

two posting lists generates Document 2, which we already have, and Document 3. Since we now have $|R| = 2$ we exit the loop and return the results to the user.

It should be clear that bottom-up performs well if there are enough documents that match the query well, requiring minimal relaxation. In particular, it is optimal if there are at least k documents at relaxation cost 0 (for instance, it would be optimal if we were interested in the top-1 document in our example). When that is not the case, it can be expensive since it needs to reinitiate the query at higher levels of relaxation from the beginning of the posting lists—since the positing lists for the higher levels have all the documents in their subtree, bottom-up keeps rediscovering the same solution over and over. Figure 4 shows 8 cursor movements (calls to either getNext() or fwdBeyond()), from which 6 are used to find Document 2.

(2) Top-down search. We start at $\ell = L$, the most general level available. If there are k or less documents at the current level ℓ, we are done since specializing further will not help. Otherwise, it is possible to specialize and still obtain k documents with better scores. So we specialize the query by decrementing ℓ, e.g., to level $\ell - 1$, after we have seen k documents at level ℓ. We only call getNextLevel() (line 6) to specialize the query if we have already seen k results although levelDone is false. However, unlike in the bottom-up case, we do not have to abandon the results R that have been computed so far. We just apply a post filtering to R to realize the specialization: this corresponds to setting $R = R \cap \text{docs}(S(b_{\ell-1}))$. An important benefit is that the budgeted relaxation search for $S(b_{\ell-1})$ need not begin at the very start of the posting lists, and may use information gleaned at the more general level to help plan the query processing at the more specific level.

Figure 5 shows the behavior of the top-down search for our running example of obtaining the top-2 documents for query (University Ave., Pizza). We start at the most generic level, $\ell = 20$, using posting lists Bay Area and Store. After seeing the first 2 documents, we have $|R| = 2$ and we enter line 6 of the algorithm to get a new level. At this point, we have Document 1, which has relaxation cost 6, and Document 2, which has relaxation cost 0 in the result set. Given these two documents in R, for a new document to enter the result set it has to have relaxation cost smaller than or equal to 6. We can then safely specialize the level down to $\ell = 10$ since

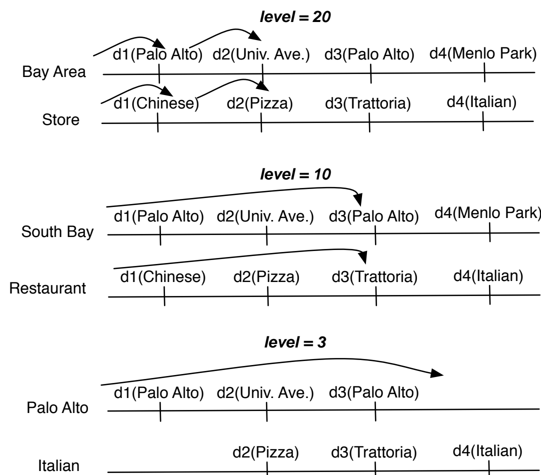

Figure 5: Top-down search example.

any document that is not part of either the South Bay or the Restaurant subtrees would have relaxation cost bigger than 6 and would not be part of the result set.

We can then proceed with posting lists South Bay and Restaurant. Moreover, we can start processing in these posting lists at *docid* 3, since we already processed all documents with *docid* smaller than 3. That can be accomplished by using the fwdBeyond() method when initializing the South Bay and Restaurant posting lists. We then process Document 3, which has relaxation cost 3, adding it to the result set to replace Document 1 in the result heap. Then we again enter line 6 of the algorithm with $|R| \geq 2$. Now the biggest relaxation cost in the result set is 3 and we know that for a new document to enter the result set its relaxation cost must be at most 3. We can then safely go to $\ell = 3$, using posting lists Palo Alto and Italian, since we know that any document outside these two subtrees would have relaxation cost above 3. Moreover, we can start at *docid* 4 at the new level. Since the posting list for Palo Alto does not have any document with *docid* ≥ 4 we are done and can return the documents in our result set to the user.

It is clear that top-down performs quite well if a fair amount of relaxation is needed to obtain k documents. Moreover, top-down is more incremental than bottom-up since it allows us to reuse the documents already in the result set when going to a more specialized level. In this simple example, top-down was done with 7 cursor movements (i.e., 7 calls to either getNext() or fwdBeyond()).

(3) Binary search. We start at the middle level, i.e., $L/2$. Depending on whether there are enough documents at the current level ℓ, we either move up or down the levels, as in a normal binary search. Figure 6 shows how the binary search would work for our running example. It starts at the middle level with $\ell = 10$ and from then on behaves similarly to top-down. Note that after having seen the first two documents, $|R| = 2$ and we enter line 6 of the algorithm. However, at this point, just like for the top-down algorithm, no further specialization is possible without potentially missing results. Therefore, we keep using the same posting lists until we see Document 3, at which point specialization becomes possible.

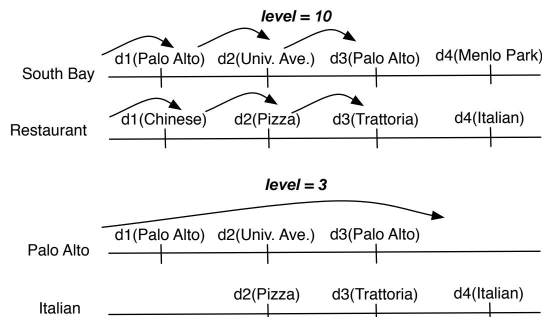

Figure 6: Binary search example.

5.3 Budgeted relaxation search

In this section we focus on the budgeted variant of relaxation search: find all documents with relaxation cost $\leq b$ at minimal retrieval cost. Recall that budget b defines a simplex $S(b)$ containing document docs($S(b)$).

For a set of grid points G, let lca(G) denote their least common ancestor, defined as the coordinate-wise least common ancestors in the corresponding trees (e.g., if $G = \{g, g'\}$ with $g = (t_1, \ldots, t_m)$ and $g' = (t'_1, \ldots, t'_m)$, then $\text{lca}(G) = (\text{lca}(t_1, t'_1), \ldots, \text{lca}(t_m, t'_m))$). For example, in Figure 3, we have lca($S(4)$) = (Palo Alto, Restaurant) and lca($S(10)$) = (Bay Area, Store). Note that by definition

$$\text{docs}(S(b)) \subseteq \text{docs}(\text{lca}(S(b)).$$

Therefore, an easy way to obtain docs($S(b)$) is to send lca($S(b)$) as a "query" to the underlying index. For example, in Figure 3, to access documents in docs($S(4)$), the query would be "Palo Alto AND Restaurant." However there are other ways of obtaining docs($S(b)$) as well, such as to issue two separate retrieval queries, say "Palo Alto AND Italian" and "University Ave. AND Restaurant" (which jointly cover all '×' marks in the simplex region $S(4)$), and then take the union of the results.

The original one-query approach has the drawback of being less selective, but the two-query approach incurs redundant processing (i.e., Italian restaurants in University Ave. are retrieved twice)[4]. Which option is cheaper? The answer depends on the joint distribution of documents in taxonomy nodes, making this a query optimization issue. Figure 7 shows a more general case of multiple rectangular queries that jointly cover a simplex and partially overlap—there are regions in the diagram covered once, twice, and three times by this set of queries, and other regions are covered once even though they lie outside the simplex of interest.

To choose among the various possible plans, cost-based query planning may be used. We assume query costs can be estimated on the fly based on statistics gathered thus far in query processing (if any). The availability of reliable statistics depends on the outer search method used (Section 5.2). In top-down evaluation, fairly good statistics are available relatively early, since the most general posting lists are scanned first. On the other hand, in bottom-up evaluation no statistics are available when moving to a higher level. Binary search has good statistics to work with some of the time, depending on the exact search progression.

[4]Of course, already retrieved documents can be cached by the engine, so that they are not scored again.

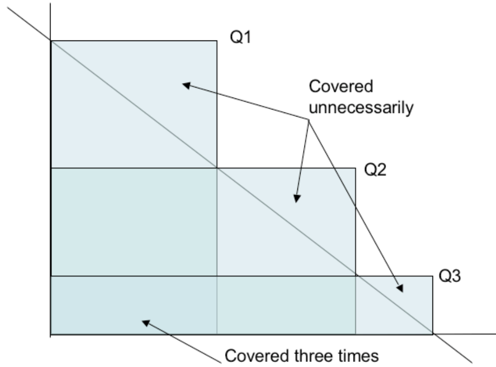

Figure 7: Three queries cover a simplex.

Note that the statistics needed for query planning have to do with the joint distribution of documents into nodes on the relaxation paths. Relaxation paths are query-specific. Gathering and storing accurate statistics on the full cartesian product of taxonomy nodes is unlikely to be feasible, especially given that many real-world taxonomies have tens of thousands of nodes. For this reason, in this paper we focus on adaptive query processing approaches that work with query-specific statistics gathered on the fly.

We describe our query planning algorithm for the case of two taxonomies, which is based on dynamic programming and is guaranteed to find the cheapest plan, in Section 5.3.1. This algorithm is independent of the model used to generate costs and if they are generated on the fly or estimated, the algorithm can still benefit from this information. In Section 5.3.2 we present a planning algorithm that finds an approximately good plan in the general case of more than two taxonomies. For situations in which no reliable statistics are available (e.g., bottom-up evaluation), our query planner opts for a single-query plan (i.e., the query $\text{lca}(S(b))$).

5.3.1 Query planning with two taxonomies

With two taxonomies, queries are points in a two-dimensional plane. A particular query (x, y) returns a set of documents; we refer to this set as $\text{docs}(x, y)$. Observe that if $x' \leq x$ and $y' \leq y$ then $\text{docs}(x', y') \subseteq \text{docs}(x, y)$. Thus, for any query (x, y), we may draw the rectangle whose corners are $\{(x, y), (x, 0), (0, y), (0, 0)\}$ to indicate the set of queries that are subsumed by the query (x, y). To generate all possible objects within a particular budget, we must simply select a set of rectangles that cover the simplex. Each query may be annotated with the cost $C(x, y)$ of performing the query (x, y), and we may then ask for the minimum-cost cover for a particular simplex; this will represent the optimal solution to the budgeted relaxation search problem whose budget corresponds to the given simplex.

THEOREM 1. *There is an efficient algorithm to solve the query planning problem with two taxonomies.*

PROOF. For a fixed simplex $S(b)$, let $(x, S(b, x))$ denote the point at which x intersects the diagonal face of the simplex, and let $B(x_0)$ denote the cost of the best cover of those points of the simplex with $x \geq x_0$. Let $\text{next}(x)$ denote the first x-axis tick mark strictly greater than x. Then,

$$B(x_0) = \min_{x \geq x_0} C(x, S(b, x)) + B(\text{next}(x)).$$

Filling in this dynamic program requires time proportional to the number of points in the simplex. The final solution is simply $B(0)$. □

5.3.2 General query planning

Although the query planning problem for two taxonomies has a polynomial time solution, the exact optimization becomes NP-hard even with three taxonomies. After presenting the NP-hardness reduction we turn to approximation algorithms, and show that a simple greedy approach can achieve a good approximation.

THEOREM 2. *The query planning problem with three taxonomies is NP-hard.*

PROOF SKETCH. For the purposes of the reduction, consider the following, simplified version of the problem. There are three taxonomies T_1, T_2, T_3 and each document is associated with a node from each of these taxonomies. Let the relaxation cost of each document be b^*; thus each document is a point on the boundary of the three-dimensional simplex $S(b^*)$. Without loss of generality we need to consider only the query points with relaxation cost at least b^*. Note that a query (x, y, z) returns all documents in $\text{docs}(x, y, z)$ at a cost $C(x, y, z)$ and the projection of this three-dimensional query point onto the simplex $S(b^*)$ yields a triangle.

Now, the query planning problem is identical to the following geometric set cover problem. Given $\mathcal{X}$, a set of n points in $\mathbf{R}^2$ (the documents) and a set $\mathcal{T} = \{t_1, \ldots, t_m\}$ of triangles (the queries), with triangle t_i having cost c_i, select $\mathcal{S} \subseteq \mathcal{T}$ of triangles of minimum total cost that covers all of the points in $\mathcal{X}$. Furthermore, we set the costs c_i so that all triangles under consideration have the same size.

This problem is a constrained version of set cover, and has been previously studied under the restriction that each t_i is an axis-parallel rectangle (instead of a triangle as it is here) and shown to be NP-hard by Fowler et al. [19]. However, their proof does not require the covering sets to be axis-parallel rectangles. In fact, it can be easily amended to have the sets t_i consist of equal-sized triangles, as is our case. Thus, the query planning problem is NP-hard, even with three taxonomies. □

On the other hand, the geometric set cover problem is well studied and yields non-trivial approximation algorithms. Let n be the total number of documents that have low enough relaxation cost. Using the standard greedy set cover algorithm we can obtain an $O(\log n)$ approximation to the query planning problem, regardless of the total number of taxonomies. However, since the problem has a nice geometric structure it is possible to obtain better approximation algorithms. In the case of d taxonomies, the set of triangles (when $d = 2$) and simplices (when $d > 2$) has VC-dimension of $O(d)$. Therefore, the randomized covering technique of Bronnimann and Goodrich [7] achieves an $O(kd \log k)$ approximation where k is the size of the optimal solution; note that when only a small number of database queries suffices, this is much smaller than $\log n$. Recently, Clarkson and Varadarajan [10] showed how to compute an $O(k \log \log k)$ approximation algorithm to this problem in the case of $d = 3$, i.e., three taxonomies. While their algorithm improves the cost guarantees, its unlikely to be competitive on real data due to its complexity.

6. EXPERIMENTS

In this section we describe a set of preliminary experiments to evaluate the proposed algorithms. We use a synthetic data set so that we can vary taxonomy characteristics such as depth, fanout, and multiplicity and a real-world data set. For these experiments we modified the Lucene open source text indexer (`lucene.apache.org`) to support relaxation search over multiple taxonomies. The primary goal of our experiments is to evaluate the relative performance of our algorithms against the baseline. To make the presentation simpler, we do not present results involving static score—our results consider only the keyword and taxonomy restrictions of the query.

We sought to evaluate the intrinsic performance properties of the various search algorithms, independent of low-level issues such as posting list compression, memory and disk speed, and caching. Hence, as done in [17], we use the total number of cursor movements as our performance metric, i.e., the total number of posting entries accessed by calls to getNext() and fwdBeyond() to answer the query. (The bookkeeping overhead of our algorithms is minimal; running time is dominated by cursor-based traversal of index posting lists.)

The *baseline algorithm*, against which we evaluate our new family of algorithms, is one that retrieves documents that satisfy the textual portion of the query and post-processes such documents using the taxonomy metadata. The post-process is done during the traversal of the posting lists, as documents are inserted into the heap that stores the top k results. This algorithm constitutes a direct application of standard IR processing to our context, and hence it does not exploit the hierarchical taxonomy structure during query processing. The baseline algorithm is as follows.
(1) Identify documents that match the keyword portion of the query (keyw(Q)) in the usual way using the text index.
(2) For each document d satisfying keyw(Q), lookup d's position in each taxonomy T and use this information to compute d's overall score.
(3) Retain the top k documents in a heap.

These three steps are executed during posting list traversal. For each document d returned by step (1), we execute steps (2) and (3). The baseline algorithm is equivalent to a variant of our top-down policy that never "moves down," i.e., it always stays at level L and never accesses taxonomy posting lists corresponding to non-root nodes.

Our synthetic taxonomies are balanced trees of varying fanout and depth. Each taxonomy has fixed depth d and fanout f, and contains every document. The documents are distributed among the leaves uniformly at random. The parameters we vary are the following.

Number of taxonomies	1–4
Fanout of taxonomies	2–8
Depth of taxonomies	4, 8
Selectivity of keywords	1.0–0.01
Number of results (k)	10, 100, 1000

6.1 Processing with one query per level

Recall that our framework for processing relaxation queries consists of two pieces: first, a scheme for moving up and down the levels of relaxation (Section 5.2); and second, an algorithm for producing a query plan to scan relevant documents at each level of relaxation (Section 5.3). In this section we simplify the second problem by assuming that in all cases, the relevant documents at each level are simply produced by a single query, even if this query must be sufficiently broad to include some overly-general documents as well. We fix $k = 10$ and examine four search algorithms: baseline, bottom-up, top-down, and binary search. Later, in Section 6.2, we will extend our scope to study the impact of scanning a level by applying multiple distinct queries, each covering a piece of the overall space.

Figure 8 shows the performance of the four algorithms, when the selectivity of the textual portion of the query is 1 (i.e., either all documents contain the requested keywords or the query did not specify any keywords). We vary the following: taxonomy depth (4 or 8), taxonomy fanout (2 or 6), and number of taxonomy restrictions in the query (x-axis of each plot). The y-axis of each each plot shows the number of cursor movements, normalized with respect to the baseline (i.e., the baseline is always at $y = 1$). In all four plots, we see that with only one taxonomy restriction, all of our algorithms dramatically outperform the baseline algorithm. The reason is that they quickly zero in on a leaf node of the single taxonomy that contains the top 10 results, whereas baseline scans all documents. For depth = 4 and fanout = 2, the performance of non-baseline algorithms is almost identical, even with taxonomy restrictions = 4. However, the performances start to diverge for greater depths and higher fanout. As the number of taxonomy restrictions grows, the performance of the non-baseline algorithms degrades and at some point becomes no better than that of baseline (in some cases it is worse). Here, full relaxation in every taxonomy is required in order to find 10 results, due to the random and independent assignment of documents to taxonomy nodes in our synthetic data. In such cases none of our algorithms can improve upon baseline, which simply scans the posting lists of the taxonomy roots.

The bottom-up algorithms performs very poorly as the number of taxonomies grows, because it wastes significant effort before converging on the right level (i.e., the root level). Binary search converges somewhat faster. Top-down never performs worse than baseline (since baseline is a degenerative case of top-down), and sometimes performs significantly better.

Figure 9 shows how performance is affected by making the textual portion of the query selective. The baseline algorithm exploits selective keywords to narrow its search. For queries with unselective keywords (0.1 or higher), all the algorithms outperform or match baseline. For queries with selective keywords, baseline outperforms or matches the other algorithms. Recall that in this section we configure our algorithms to always issue a single query per level. This restriction handicaps the ability to exploit the taxonomies, so when the keywords are even mildly selective, no benefit is derived from early taxonomy-driven filtering. Next we study the use of multiple queries per level which improves the ability to exploit taxonomy restrictions early in the processing.

6.2 Multiple queries per level

In this section we apply our algorithms to control the level of relaxation. At each level, we show results for using a single query, as in the previous section, or multiple queries. To determine the appropriate set of multiple queries to submit, we employed the dynamic programming technique described in Section 5.3.

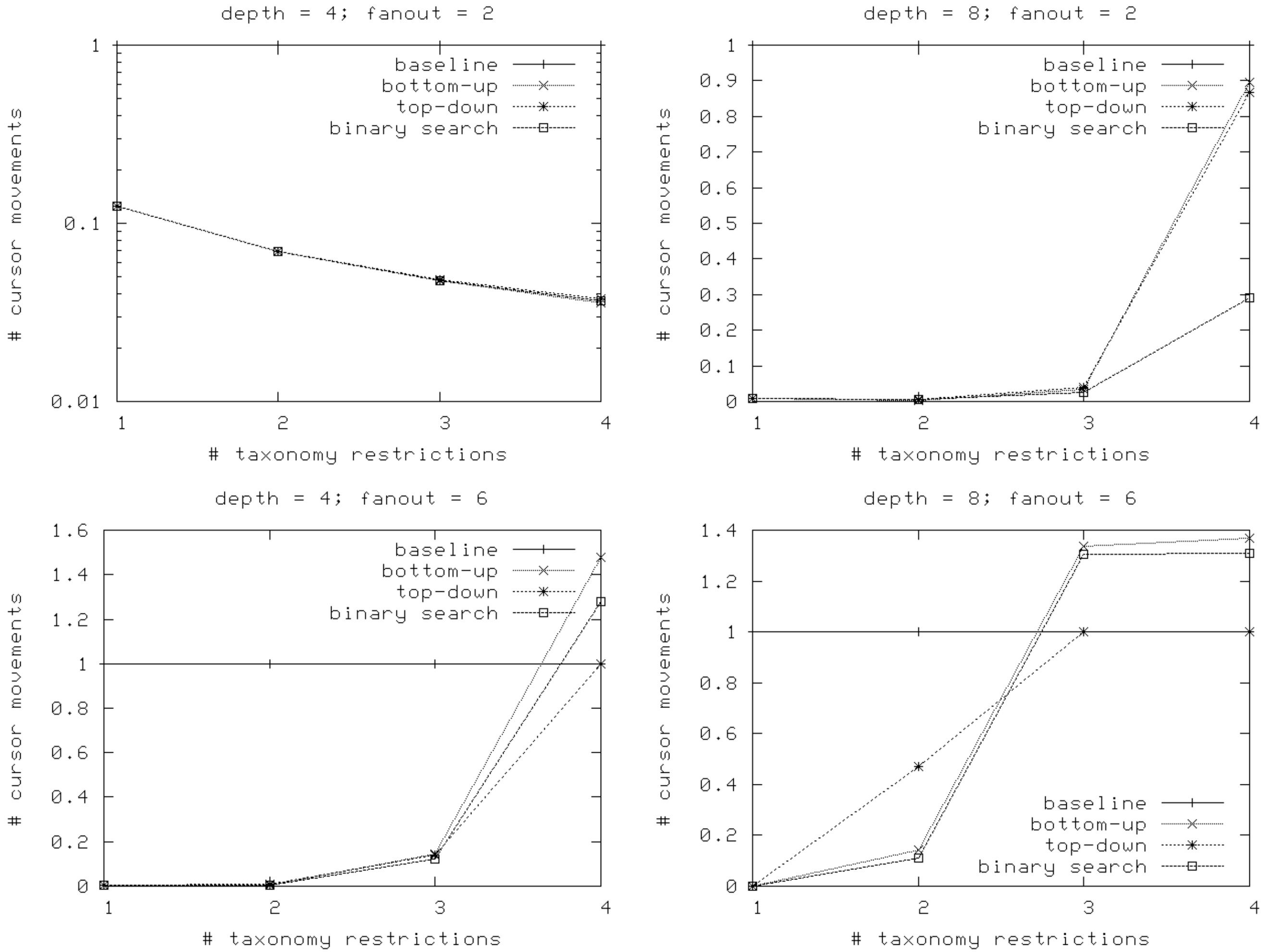

Figure 8: Performance on a low-selectivity text query (all documents contain the query term) on multiple taxonomies. The y-axis is the number of cursor movements, relative to the baseline.

The dynamic program requires the cost of each query. We do not have this cost available. As our algorithm is forced to replan each time the level changes, it is natural to maintain dynamic statistics. We keep for each point of the full simplex the count of documents and the largest *docid* examined so far. Based on this, we estimate the cost of performing each query. When no count is available, as for example when we are generalizing upwards, we treat all queries equally. However, to focus on the difference between single and multiple queries, we ask the algorithm to return multi-query solution when no information indicates otherwise. For example, if the algorithm is asked to cover two taxonomies at relaxation budget 2, in the absence of other information it will return three queries: (0,2), (1,1), and (2,0).

Figure 10 shows the performance of the binary search algorithm for both single and multiple queries. The benefits of multiple queries for taxonomies with larger depths and higher fanout can be clearly seen from this figure. As an illustrative example, consider two taxonomies, each with fanout 8, and a relaxation budget of 2. The query $(2, 2)$ will access two postings, each of which is a factor of 64 times larger than the leaf postings for that taxonomy. Instead, the queries $(1, 2)$ and $(2, 1)$ will access only one large posting and one posting that is eight times smaller; thus, each query will touch roughly an eighth as much data and only two such queries are required to cover the space. This example shows why a situation of high fanout may be particularly amenable to multiple queries.

Algorithm	Number of results	
	k=10	k=100
baseline	11277	
bottom-up	819	1582
top-down	61	242
binary search	62	242

Table 1: Avg. number of cursor movements.

6.3 Experiments on real-world data

We evaluated the performance of our algorithms on the Reuters dataset RCV1 [28] that contains 810K English language news articles from the period of Aug 20, 1996 to Aug 19, 1997. The total uncompressed size of the data is about 2.5GB. Each document in this collection is classified into two taxonomies. The first is the “industry” taxonomy that

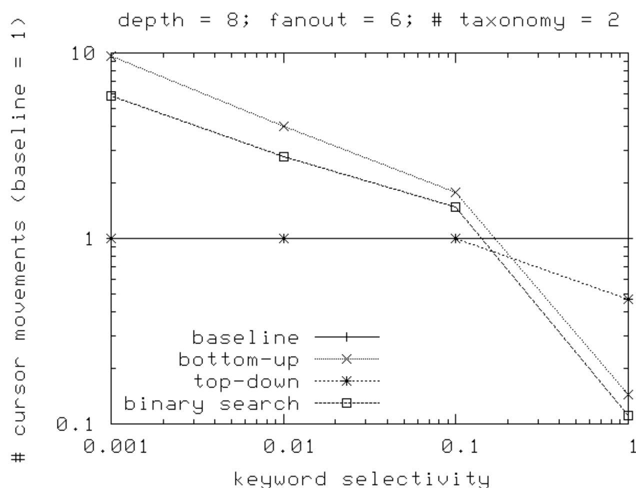

Figure 9: Performance wrt keyword selectivity of query.

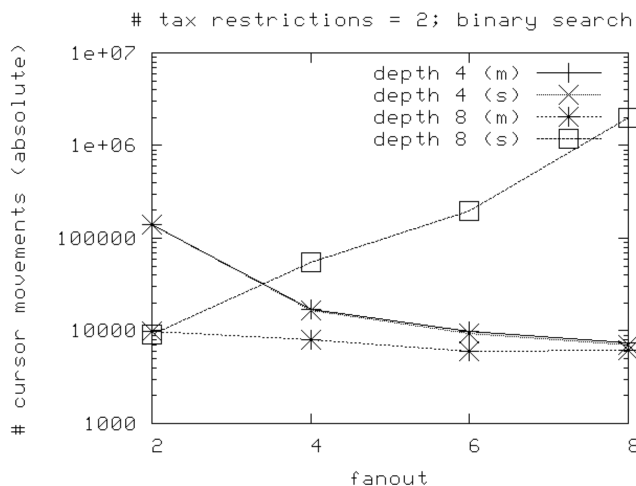

Figure 10: Performance wrt single (s) versus multiple (m) queries per level.

has 996 leaf nodes; the maximum depth of the taxonomy is 7. The second is the "date" taxonomy that has 1140 leaf nodes; the top level in this taxonomy represents the year, the second level represents the month, and the third level represents the day.

For this dataset we picked 1000 random queries consisting of pairs of nodes from the first and second taxonomies. To demonstrate the impact of the algorithms, only nodes at depth five or greater from the first taxonomy were considered. Table 1 shows the number of cursor moves for all the queries for each algorithm, varying the number of results requested (k).

Top-down performs extremely well, especially since the second taxonomy is quite shallow. Binary search is on par with top-down and may be more robust for deeper taxonomies. Both top-down and binary search outperform the baseline and bottom-up by orders of magnitude.

7. SUMMARY

We studied relaxation search in taxonomies by proposing a general framework. We integrated this framework into the IR context to obtain a novel document retrieval model, which has several and diverse applications, ranging from product search to online advertisement. We proposed efficient indexing and query processing algorithms to implement this new search paradigm and extensively evaluated our algorithms on both synthetic and real-world data. Our experimental results show that (1) this novel search paradigm is viable and (2) the algorithms we proposed outperform the standard IR solutions for the problem by orders of magnitude. We also reported theoretical results for the budgeted query processing problem, showing that in the general case it admits efficient approximation algorithms with provable guarantees.

In our work we have assumed that the taxonomy and the weights are provided as input to the framework. This works well in many practical settings. However, when the taxonomy is not well-specified or when it is not clear how to chose the weights, learning the taxonomy and defining reasonable weights become interesting research questions themselves.

Acknowledgments

We thank the anonymous referees who provided many helpful suggestions.

8. REFERENCES

[1] R. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. In *Proc. 13th ICDE*, pages 232–243, 1997.

[2] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *Proc. 18th ICDE*, pages 5–16, 2002.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[4] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proc. 30th ACM STOC*, pages 161–168, 1998.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *WWW/Computer Networks*, 30(1-7):107–117, 1998.

[6] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Y. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. 12th ACM CIKM*, pages 426–434, 2003.

[7] H. Bronnimann and M. T. Goodrich. Almost optimal set covers in finite VC-dimension. In *Proc. 10th ACM SoCG*, pages 293–302, 1994.

[8] D. Burdick, P. M. Deshpande, T. S. Jayram, R. Ramakrishnan, and S. Vaithyanathan. OLAP over uncertain and imprecise data. In *Proc. 31st VLDB*, pages 970–981, 2005.

[9] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *Proc. ACM SIGMOD*, pages 277–288, New York, NY, USA, 2006. ACM.

[10] K. L. Clarkson and K. Varadarajan. Improved approximation algorithms for geometric set cover. In *Proc. 21st ACM SoCG*, pages 135–141, 2005.

[11] W. F. Cody, J. T. Kreulen, V. Krishna, and W. S. Spangler. The integration of business intelligence and

knowledge management. *IBM Systems Journal*, 41(4):697–713, 2002.

[12] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Diamond in the rough: Finding hierarchical heavy hitters in multi-dimensional data. In *Proc. ACM SIGMOD*, pages 155–166, 2004.

[13] R. Fagin, R. Guha, R. Kumar, J. Novak, D. Sivakumar, and A. Tomkins. Multi-structural databases. In *Proc. 24th PODS*, pages 184–195, 2005.

[14] R. Fagin, P. Kolaitis, R. Kumar, J. Novak, D. Sivakumar, and A. Tomkins. Efficient implementation of large-scale multi-structural databases. In *Proc. 31st VLDB*, pages 958–969, 2005.

[15] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.

[16] D. Florescu, D. Kossmann, and I. Manolescu. Integrating keyword search into XML query processing. *Computer Networks*, 33(1-6):119–135, 2000.

[17] M. Fontoura, V. Josifovski, E. Shekita, and B. Yang. Optimizing cursor movement in holistic twig joins. In *Proc. 14th ACM CIKM*, pages 784–791, 2005.

[18] M. Fontoura, E. J. Shekita, J. Y. Zien, S. Rajagopalan, and A. Neumann. High performance index build algorithms for intranet search engines. In *Proc. 30th VLDB*, pages 1158–1169, 2004.

[19] R. J. Fowler, M. Paterson, and S. L. Tanimoto. Optimal packing and covering in the plane are NP-complete. *IPL*, 12(3):133–137, 1981.

[20] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 2000.

[21] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *DMKD*, 1(1):29–53, 1997.

[22] D. Gruhl, L. Chavet, D. Gibson, J. Meyer, P. Pattanayak, A. Tomkins, and J. Y. Zien. How to build a WebFountain: An architecture for large-scale text analytics. *IBM Systems Journal*, 43(1):64–77, 2004.

[23] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *JASIST*, 54(8), 2003.

[24] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *Proc. 29th VLDB*, pages 850–861, 2003.

[25] H. V. Jagadish, L. S. Lakshmanan, and D. Srivastava. What can hierarchies do for data warehouses? In *Proc. 25th VLDB*, pages 530–541, 1999.

[26] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proc. 31st VLDB*, pages 505–516, 2005.

[27] E. Kandogan, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar semantic search: A database approach to information retrieval. In *Proc. ACM SIGMOD*, pages 790–792, 2006.

[28] D. Lewis, Y. Yang, T. Rose, and F. Li. RCV1: A new benchmark collection for text categorization research. *JMLR*, 5:361–397, 2004.

[29] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *Proc. 29th VLDB*, pages 129–140, 2003.

[30] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. In *Proc. 10th WWW*, pages 396–406, 2001.

[31] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[32] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *IPM*, 31(6), 1995.

[33] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann, 1999.

[34] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *Proc. ACM SIGMOD*, pages 537–538, 2005.

[35] K.-P. Yee, K. Swearingen, K. Li, and M. Hearst. Faceted metadata for image search and browsing. In *Proc. ACM CHI*, pages 401–408, 2003.

[36] X. Zhou, J. Gaugaz, W.-T. Balke, and W. Nejdl. Query relaxation using malleable schemas. In *Proc. ACM SIGMOD*, pages 545–556, 2007.