XM-Tree, a new index for Web Information Retrieval

Claudia Deco, Guillermo Pierángeli, Cristina Bender
Departamento de Sistemas e Informática
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
(2000) Rosario, Argentina
{deco, bender}@fceia.unr.edu.ar, guillepier@yahoo.com

and

Nora Reyes
Departamento de Informática,
Universidad Nacional de San Luis
(5700) San Luis, Argentina
nreyes@unsl.edu.ar

ABSTRACT

Web Information Retrieval is another problem of searching elements of a set that are closest to a given query under a certain similarity criterion. It is of interest to take advantage of metric spaces in order to solve a search in an effective and efficient way. In this article, we present an extension of the M-Tree index, called *XM-Tree*, in order to improve search results. This index allows dynamic insertion of new data, reduces search costs using pruning and precalculated distances, and uses a tolerable amount of space, which makes this index apt for the extensive and dynamic Web. The proposed extension indexes Web documents, uses L_2 as indexing distance and L_∞ as similarity criterion to solve queries. We also present experiments validating the results.

Keywords: Metric Spaces, Similarity Searching, M-Tree, XM-Tree

1. INTRODUCTION

Searching is a fundamental problem in computer science, which is present in almost every computer application. The search operation has been traditionally applied to "structured data". With the evolution of information and communication technologies, unstructured repositories of information have emerged, such as the Web. New data types such as free text, images, audio and video have to be queried, but their structuring is very difficult (either manually or computationally) and restricts beforehand the types of queries that can be posed later. This scenario requires more general search algorithms and models than those classically used for simple data. There is a need for searching database elements which are similar or closed to a given query element. Similarity is model with a *distance function* that satisfies the triangular inequality, and the set of objects is called a metric space. In some applications, the metric space turns out to be of a particular type called vector space.

A lot of work tries to achieve the goals of reducing the number of distances evaluations and the amount of I/O performed, in general around the concept of building an index, a data structure designed to reduce the amount of distance evaluations at query time. [1] presents a unified framework that describes and analyzes existing solutions to this problem, and there are two main techniques for indexing: one is based on pivots and another is based on Voronoi partitions. Within these techniques, we propose in this work, an extension of M-Tree, to index documents and speed up web searches discarding as much irrelevant objects as possible.

The rest of this paper is organized as follows: Section 2 presents basic concepts of Information Retrieval and Metric Spaces. Section 3 presents related work. Section 4 proposes the XM-Tree as an extension of M-Tree. Section 5 presents the experimental results. Finally, conclusions are presented.

2. BASIC CONCEPTS

Information Retrieval

An Information Retrieval problem is, given a set of documents and a query, to determine the subset of relevant documents to the query [2]. An approach to solve this problem is to use search engines based on traditional Information Retrieval techniques. These search engines locate information in a set of documents, through two steps: indexing and retrieval. The documents are indexed by the terms that contains. The process of generation, construction and storing document representations is called indexing and as a result, we obtain inverted indexes. An inverted index allows fast access to the list of documents that contains a specific term. For this purpose, it maintains a register for each term in the document collection, which in its simplest form consists in the term name and the list of documents where that term occurs. Each term occurrence in the inverted index is called a *posting*, and the list of postings of a term is called a *posting list*. For retrieval, the search engine uses this inverted file to look up which documents contain the query words, and obtains the posting list for each one. These posting lists are then merged. This process depends on the engine. For example, a boolean engine requires the query be formulated using boolean operators between words. Merging is then straightforward as the documents sets are combined using the appropriated set operators.

The typical indicators to measure the effectiveness in information retrieval are called Precision and Recall. *Precision* is the ratio of the number of relevant documents retrieved to the total number of documents retrieved. *Recall* is the ratio of the number of relevant documents retrieved to the total number of relevant documents in the collection. Both are usually expressed as a percentage. An engine achieves a good performance when it maximizes both values. This is, when it retrieves most relevant documents available in the collection along with the least amount of irrelevant documents.

One way to represent documents is the *vector space model*, in which documents are represented with vectors of words. The very common terms, articles, and pronouns are not considered in these vectors. In addition, verbs, nouns and adjectives, are reduced to a canonical form with a *stemming* process. The document is represented as a vector in the Euclidean space with the resulting set of words. Each canonical term represents an axis in this space. The i-th component of vector d is 1 if term t_i occurs in document d, or 0 otherwise. Queries are represented in the same way. The similarity between a query q and a document d can be measured by computing the dot product of vectors q and d: $sim(q, d) = \Sigma\ q_i \times d_i$. This process allows us to return a sequence of documents ranked by their relevance to the query.

The problem with this representation is that it does not capture the fact that some terms might be more important than others in a single document. The solution is to represent the weight of the word in the document at each component. This weight may be the number of occurrences, but this representation gives preference to larger documents. This could be solved using the *TF scheme*, or term frequency scheme, which normalizes the vectors by dividing each component by the vector length. Thus, the value tf_i of each component d_i is the frequency of the term t_i in document d. The TF scheme does not consider the distribution of the term in the whole collection. This is solved with the concept of inverse document frequency: $idf_i = log\ (N / n_i)$, where N is the total number of documents in the collection and n_i is the number of documents in which i-th term occurs. In practice, both measures are combined into a schema called TF/IDF: $w_{ij} = tf_{ij}$ x idf_i w_{ij}, where w_{ij} is the value that is assigned to i-th component of the document j.

Metric Spaces

The set X denote the universe of valid objects. The function d: $X \times X \rightarrow R^+$ denotes a measure of "distance" between objects.

The distance function d has the following properties: positiveness ($d(x, y) \geq 0$), symmetry ($d(x, y) = d(y, x)$), reflexivity ($d(x, x) = 0$), strict positiveness ($\forall x, y \in X, x \neq y \Rightarrow d(x, y) > 0$), and triangular inequality ($d(x, z) \leq d(x, y) + d(y, z)$). Then the pair ($X$, d) is called a *metric space*. A finite subset U of X, of size $n = |U|$, is the set of objects where we search. U will be called the dictionary or database. The types of queries of interest in metric spaces are *Range query* and *k-Nearest neighbours query.*

Range query $(q, r)_d$ retrieves all elements which are within distance r to q, this is $\{ u \in U \mid d(q, u) \leq r \}$.

k-Nearest neighbours query $NN_k(q)$ retrieves the k closest elements to q in U. This is, it retrieves a set $A \subseteq U$ such that $|A| = k$ and $\forall u \in U, v \in U - A, d(q, u) \leq d(q, v)$.

In this work, we propose to use *Strict range query* $(q, r)_d$. This query retrieves all elements which are at a distance less than r to q, this is $\{u \in U \mid d(q, u) < r\}$.

If elements of the metric space (X, d) are indeed tuples of real numbers then the pair is called a *vector space*.

A k-dimensional vector space is a particular metric space where objects are identified with k real-valued coordinates $(x_1,.., x_k)$. There are a number of options for the distance function to use, but the most widely used is the family of L_s distances, defined as

$$L_s((x_1, ..., x_k), (y_1, ..., y_k)) = (\Sigma\ |x_i - y_i\ |^s)^{1/s}$$

The L_1 distance accounts for the sum of the differences along the coordinates, it is also called "block" distance. The L_2 distance is the "Euclidean" distance, and it corresponds to the notion of spatial distance. The other most used member of the family is L_∞, which corresponds to taking the limit of the L_s formula when s goes to infinity. The result is that the distance between two objects is the maximum difference along a coordinate:

$$L_\infty((x_1, ..., x_k), (y_1, ..., y_k)) = \max |x_i - y_i\ |.$$

Indexes for metric spaces allow the effective and efficient retrieval of objects. It is effective because the results have a high degree of accuracy by properties of space and index. It is efficient because the construction of indexes is intended to reduce the number of computations. These two properties are highly desirable in a search engine. For this, the use of metric spaces is an interesting alternative in indexing and searching processes.

A unified model of indexing algorithms for metric spaces is presented in [1]. All the indexing algorithms partition the set U into subsets. An index is built to determine the candidate subsets where relevant elements to the query could appear. There are two main types of indexing algorithms: pivot based and Voronoi based algorithms (or based on compact partitions). Pivot based algorithms consider the distances between an element and k preselected points, called "pivots", and maps the metric space on R^k using L_∞ distance. Voronoi based algorithms partition metric space considering the proximity to a set of points called centres. Pivot based algorithms need much more space to store k classes than Voronoi based algorithms using the same number of partitions. If pivot based algorithms have the necessary memory and use the optimal number of pivots, then their search cost is better than Voronoi based algorithms. But this does not occur if the intrinsic dimension of the metric space grows, because it is more difficult to search in it.

In this work, we use web pages represented with word vectors. These vectors have a very large amount of components, so the intrinsic dimension of the metric space could be high. Considering this, the Voronoi based algorithms seem to be better. In addition, the web is very dynamic, so the proposed index should enable to add data to the index. Because of this, we propose to extend the M-Tree index, which improves previous algorithms, allows insert and effectively remove elements from a tree.

M-Tree

The access method *M-Tree* [3] is proposed to organize and search large data sets. Experimentations demonstrate that M-Tree performs reasonably well in high-dimensional data spaces and growing data sets. M-Tree is a paged, balanced and dynamic tree, and it allows dynamic insertion and deletion[1] of data, and does not require periodical reorganizations after these operations. M-Tree nodes are fixed-size and they allow a maximum of m child nodes. Leaf nodes of any M-Tree store all indexed

[1] Actually, the real deletions can only be made on leaves, in case of routing object they are only setting as "deleted".

database objects, whereas internal nodes store the so-called *routing objects*. A routing object is a database object to which a routing role is assigned by a specific *promotion* algorithm.

Each node N of the M-Tree corresponds to a region of the indexed metric space (X, d). The region of node N is

$$Reg(N) = \{ O \in X \mid d(O_r, O) \leq r(O_r) \}$$

where O_r is the routing object of node N and $r(O_r)$ is its *covering radius*. All the objects in the sub-tree rooted at N are guaranteed to belong to $Reg(N)$, thus their distance from O_r does not exceed $r(O_r)$. Entries in the *internal node* N are: a routing object O_r; the covering radius $r(O_r)$; an associated pointer $ptr(T(O_r))$, which references the root of a sub-tree $T(O_r)$ called the *covering tree* of O_r; anda distance from its parent object $P(O_r)$, that is the routing object which references the node where the O_r entry is stored. This distance is not defined for entries in the root of the M-Tree. The *leaf nodes* are quite similar to routing objects, but no covering radius is needed, and the pointer field stores the object identifier in the database.

In [3] are presented two algorithms for similarity search: range query and k-nearest neighbours query. Their objective is to reduce the number of accessed nodes and the number of distance computations needed to execute queries. For this purpose, all the information concerning pre-computed distances stored in the M-Tree nodes, i.e. $d(O_r, P(O_r))$ and $r(O_r)$, are used to effectively apply the triangle inequality. Although [3] treats range queries as no strict, in this work we propose to use strict range queries $(Q, r(Q))_d$, which selects all the database objects such that $d(O_j, Q) < r(Q)$.

Our *RangeSearch* algorithm retrieves all objects, which satisfy the above inequality. For this purpose, RangeSearch starts from the root node and recursively traverses all the paths, which cannot be excluded, from leading to leaf nodes with objects that satisfy the query. When the algorithm accesses objects O_r in node N, the distance between query Q and parent object O_p is computed only once. The index stores the pre-computed distances between O_r and O_p, so it is possible to prune a sub-tree without computing any new distance at all.

3. RELATED WORKS

The metric index M-Tree was improved in several subsequent versions in order to obtain faster indexing and retrieval. The *Slim-tree* [4] is a version that accelerates the indexing process with a new split algorithm based on *minimal spanning tree*, and reduces the overlap between regions with the *Slim-down* algorithm, improving the query performance. The *M^+-tree* [5] introduces a concept called *key dimension*: splits a subspace into two subspaces, called *twin nodes*, which are not overlapped. In the searching process, the key dimension is used to perform effective pruning while is not needed to compute distance between objects. *BM^+-tree* [6] extends the *M^+-tree*. It uses a rotatable binary hyperplane, instead of a key dimension, to partition the twin subspaces and to perform filtration between them. The *Density-Based Metric tree* [7], or *DBM-Tree*, minimizes the overlap between high-density nodes by relaxing the height balancing of the tree. Thus, the height of the tree is larger in denser regions. It achieves higher performance in searches because it is possible to adjust the tree according to the data distributions at different regions of the data space. The *DBM*-tree* [8] extends the previous one: each node has an associated matrix, which contains some precomputed distances between objects of current node. Making use of pre-calculated distances, construction and query costs are reduced because it increments the pruning of irrelevant elements. In [9] basic principles and experimentation results of a paged and balanced index structure, called *M^2-tree*, are presented. The proposed approach combines within single index structure information from multiple metric spaces. This allows efficiently support queries on arbitrary combinations of indexed attributes. The *XM-Tree* proposed in this work is a very particular case of *M^2-tree*, because it extends an *M-Tree* on vector spaces, adapting its algorithms for the Web search.

With regard to web search engines, the continuously growth of the web is a great challenge to the creation of fast indexes. One solution is the use of inverted indexes, employed by search engines, such as Google [10]. However, these indexes have a disadvantage: they index by single word. When the query contains several words, the inverted index retrieves all documents, which contain at least one submitted word. Then, these documents are filtered in a union or intersection process. In order to improve response time, these engines return not only documents that satisfy the search strategy, but also they return those documents, which contain any of the query terms. The indexing process of XM-Tree proposed in this work improves these results.

4. XM-TREE PROPOSAL

In this work, we propose to use properties of indices on metric spaces in order to improve search results. The analysis of indexing algorithms of metric spaces shows that M-Tree fits for web environment. We decide to extend the M-Tree because it chooses the paths by comparing the query and information stored in the index. Moreover, the M-Tree achieves speed in the search process because it excludes sub-trees, which do not contain data close to the query. The retrieved results are correct because the pruned sub-trees always contain irrelevant data, by definition of M-Tree. In order to achieve both objectives: speed and quality results, we choose an adequate representation of web pages and a convenient similarity criterion to enable the use of the M-Tree. Below, we present the choice made for this proposal.

Dictionary Generation: The dictionary is a text file intended to measure word occurrences in documents. Each entry corresponds to a term. The dimension of dictionary vector will be called *T_DIC*.

Document Representation: The documents are represented as points in a vector space, whose dimension is also T_DIC. Each axis of this space represents a dictionary entry. We use the TF schema in order to represent every document[2]. This schema is effective because documents that share very common words (contained in the dictionary), are represented by vectors that are very close at any L_s distance. In this situation, an M-Tree with an L_s distance, groups these documents in only one sub-tree. This accelerates queries that contain shared terms.

Indexing: The index used in this work is an M-Tree whose indexing distance is L_2. L_2 distance allows grouping similar documents in TF schema successfully. L_2

[2] We discarded using TF/IDF scheme because it requires for the computation of each component, the number of word occurrences in the rest of the documents. Thus, when a new document is added, all document vectors must be recalculated, and the index would not be dynamic.

is preferred to L_∞ and L_1 because it is more accurate in a vector space.

Searching: A query has one or more word and it is represented by a vector *q* with real values and dimension T_DIC. If the term *t* is in the dictionary, *t*-th component of *q* has a not null value. In other cases, components are null. The main object is to retrieve documents where all query words co-occur at the same document. The M-Tree searching algorithm retrieves all documents which frequency vectors are closer to *q*, with the L_2 norm, so we have to choose adequate values for not null components in *q*. As the frequency of a *t*-th component in different documents vectors can be quite disproportionate, it is impossible to obtain an optimal *q* because we would have to know in advance which documents are intended to retrieve. This is a problem too, if we tried to make a query by rank: what rank *r(Q)* must be computed? Given these circumstances, it was decided to use a criterion of similarity between documents and query more relaxed than L_2.

It is possible to obtain documents that contain all query terms, checking if components that correspond to those terms are not null and ignoring the remaining components. The way to solve this is by subtracting 1 to each component. If the result is 1, neither the word nor their morphological variants are contained in the document. If the result is less than 1, then the document contains the word. Then, the criterion of similarity is: to allocate 1 into elements of *q* that correspond to dictionary terms, to compute the subtraction between vector *q* and each document vector.

If all components in a resulting vector are less than 1 then that document satisfies the query. The rest of the components in the query vector, whose terms are not in the query, are null. As frequencies by definition vary between 0 and 1, and its sum is 1, these differences for terms not included in the query, do not reach the value 1 and would not affect the criterion.

This new similarity criterion between documents and queries can be formulated with the computation of the L_∞ norm between the document vector and the query vector. If the result is less than 1 then the document contains all the query terms, otherwise, is discarded. As the M-Tree search algorithm uses the same distance for indexing and searching, we should use the L_∞ as metrics to build the tree. This option is not attractive because it determines the proximity of two documents, taking into account the frequency of a single term, whose frequency difference in both documents is maximal. Because of this, with L_∞ some documents could be considered similar, even though they do not share terms.

The other alternative is to continue using L_2 distance for indexing, and L_∞ for searching. We have chosen this option by modifying the search algorithm. This approach implies L_2 cannot be used for pruning. It is required some additional information at the internal node of the root of subtree analyzed. This is also included in our proposal.

Proposal of an eXtended M-Tree: XM-Tree

XM-Tree is an index over vector spaces to search on the web. It is an extension of M-Tree. XM-Tree treats separately the indexed vectors components. These extensions are to include additional information at nodes and to make adaptations of build algorithms to maintain this information. The added information contains distances between indices data components treated separately. This information is added to preserve these measures and to use it in a possible pruning at search time. Therefore, we have an index over web pages by using L_2 as indexing distance and L_∞ as searching distance.

To do this, a covering radius vector $rv(O_r)$ is added to internal nodes. If *n* is data dimension, vectors O_r and O_j have respectively O_{ri} and O_{ji} components, and we consider a *f* distance over components space, then for each component must be $rv_i(O_r)$: $f(O_{ri},O_{ji}) \le rv_i(O_r)$ $i = 1, 2, .., n$ $\forall O_j \in T(O_r)$. Distance *f* between components is a L_s norm and it has only one dimension. Because of this, for every *s,* its value is the absolute value of the difference. Also, a distances vector $dv(O_r, P(O_r))$ is added, which have distances between routing object components O_r and parents components $P(O_r)$. Each component is: $dv_i(O_r,P(O_r)) = f(O_{ri},P(O_r)_i)$, $i = 1,2, ..., n$.

Therefore, each XM-Tree internal node has the following information

O_r	routing object
$ptr(T(O_r))$	pointer to the root of $T(Or)$
$r(O_r)$	covering radius of Or
$\boldsymbol{rv(O_r)}$	***covering radius vector of components of*** $\boldsymbol{O_r}$
$d(O_r, P(O_r))$	distance of Or from its parent
$\boldsymbol{dv(O_r, P(O_r))}$	***distances vector between*** $\boldsymbol{O_r}$ ***components and his parent***

Leaf nodes do not change. They are like in the original M-Tree. Entries are: O_j database object, $oid(O_j)$: object identifier and $d(O_j, P(O_j))$: distance of O_j from its parent.

Like in M-Tree, searching algorithm of our extension try to reduce the amount of accessed nodes and distance computations, using precomputed distances maintained on internal nodes in $rv(O_r)$ y $dv(O_r,P(O_r))$ and using triangular inequality.

XM-Tree works on data of an *n*-dimensional vector space, with an *f* distance over components space. Given a query $Q = (Q_1, Q_2, ..., Q_n)$ and a vector $rv(Q)$ of search radius with $rv_i(Q)$ components, then the Strict range query $(Q,rv(Q))_f$ searches for all database objects O_j that verify $f(O_{ji}, Q_i) < rv_i(Q)$, $\forall i = 1, 2, ..., n$.

The modified *RangeSearch* algorithm starts from the root node and traverses all not excluded sub-trees towards the leaf nodes with objects, which satisfy the query.

```
RangeSearch(node N,query_object Q,search_radius rv(Q))

1.  Let Op be the parent object of node N
2.  If N is not a leaf then
3.    For each object Or in N do
4.      If (|f(Opi, Qi) – dvi(Or, Op)| < rvi(Q) + rvi(Or)
            i =1..n) then
5.        Compute f(Ori, Qi) i =1..n
6.        If (f(Ori, Qi) < rvi(Q) + rvi(Or) i=1..n) then
7.          RangeSearch(*ptr(T(Or)), Q, rv(Q))
8.  else  // N is a leaf
9.    For each node child Oj in N do
10.     If (|f(Opi, Qi) – dvi(Oj, Op)| < rvi(Q) i=1..n) then
11.       Compute f(Oji, Qi) i =1..n
12.       If (f(Oj, Qi) < rvi(Q) i =1..n) then add oid(Oj) to
              the result set
13. End
```

At line 3, algorithm accesses O_r objects of *N* node. Distances between Q_i components and O_{pi} parent object are computed only once for i= 1... n. Distances between O_{ri} a O_{pi} are precomputed in vector $dv(O_r, O_p)$. Therefore,

we can prune sub-trees without computing other distances. The pruning is made at line 6, if the condition $f(O_{ri}, Q_i) \geq rv_i(Q) + rv_i(O_r)$ is true. In this case, to each O_j in $T(O_r)$ it is $f(O_{ji}, Q) \geq rv_i(Q)$. So, $T(O_r)$ could be discarded. At line 9, the algorithm checks leaf nodes against O_j objects. Distances between O_{pi} y Q_i components are computed only once and $dv_i(O_j, O_p)$ elements are set in the structure. In lines 4 and 10, searching in not relevant nodes are avoided, considering that if data covering radius $rv_i(O_j)$ are null: If $|f(O_{pi}, Q_{ii}) - dv_i(O_r, O_p)| \geq rv_i(Q) + rv_i(O_r)$ then $f(O_{ri}, Q_{ii}) \geq rv_i(Q) + rv_i(O_r)$. Both conditions are valid because they are particular cases of lemmas proposed by [3].

The original M-Tree *Insert* and *Split* algorithms were modified only by adding sentences for maintaining the new data: $rv(O_r)$ y $dv(O_r, P(O_r))$. Once web pages are represented with TF schema vectors, XM-Tree is built using L_2 norm as indexing distance d and L_s, norm as distance f between components. Searching algorithm is called with a radius vector of components $rv_i(Q)$ of value 1. With this, we can index with L_2 and search with L_∞. In the case of searching, we exploit the following equivalence within proposed search criterion and XM-Tree Strict range query:

$(Q, rv(Q))_{f:} : L_\infty(Q, O) < 1$
$\leftrightarrow \max\{|Q_i - O_i|\} < 1$
$\leftrightarrow |Q_i - O_i| < 1$
$\leftrightarrow f(Q_i, O_i) < 1$
$\leftrightarrow f(Q_i, O_i) < rv_i(Q) \quad \forall i=1, 2, ..., n.$

Achieving this equivalence justifies the decision to employ strict range query, since the original range query would not be useful because it would achieve the radius value $rv_i(Q)$.

5. EXPERIMENTATION

Prototype description

The system proposed in this work was implemented in C^{++} using DJGPP [11]. We built a dictionary with 1000 entries, considering morphological variants of each word. TF vectors of web pages are computed by using only visible text. For each web page, there is a file with: identifier, frequency vector, title, URL, and backup allocation. Also, Insert, Split and Search algorithms were implemented. When the algorithm starts, it reads the data file, it builds dynamically the tree and it asks for the query. Tree arity m is a fixed parameter, and users can use logical operators. AND is the default logical operator.

For Split algorithm, m_RAD and Generalized Hyperplane policies were chosen to model promote and partition algorithms. In m_RAD, "minimum sum of radii" algorithm promotes objects which it covering radius sum $r(Op_1) + r(Op_2)$ is minimal. Generalized Hyperplane assigns each $O_j \in S$ to the closest promoted object. The strategy is not balanced. We use only these two policies. A future work, is to analyze others policies.

Once the query is processed, the prototype shows data from three types of searches: searching the XM-Tree, an exhaustive search with the original query on all indexed pages, and a search that emulates an inverted index search. The interface provides information on the structure of the built XM-Tree: amount of internal nodes, amount of leaf nodes and data indexed. To analyze the search efficiency in the tree, it shows: amount and percentage of routed nodes and pruned nodes. To verify the search effectiveness in the XM-Tree, the interface shows the amount of results, and Precision and Recall indicators. After the filtering process, are shown again the amount of results, and Precision and Recall, which would be the final values achieved by the system. The search strategy that invokes the algorithm is compared with vectors of all pages using the L_∞ criterion similarity. Those pages that verify the criterion and were not returned in the search, or those that non-verified the criterion and were returned, are shortcoming in the XM-Tree and its amount is shown under the title 'mistakes'.

The exhaustive search checks if query terms appear in the text of each page. The aim is to show the effectiveness of the following proposals made in this work: scheme TF as a mechanism for pages representation, the L_∞ similarity criterion between page and query, and XM-Tree as an indexing method. The amount of different results between exhaustive and XM-Tree are showed. The amount of results obtained in the exhaustive search is considered the total number of relevant documents in the collection and it is used to compute Precision and Recall indicators. The last search simulates an inverted file index. The difference between the retrieved data by the first instance of this search and the XM-Tree search is also showed. This comparison allows verify the largest efficiency of the XM-Tree on inverted indexes. Data are sorted according to the maximum internal product with q vectors, from highest to lowest.

Experimental Results

Extensive experiments with the prototype are conducted to show system performance in a similar environment to actual Web environment. The pages that indexes the XM-Tree in this testing is a very small set compared with the total indexed Web, because its objective is only experimental. For this reason, we attempt to build it in the most representative way as possible.

In order to simulate the thematic heterogeneity that characterizes the Internet in a small sample, we retrieved real Web pages with different topics but sharing some significant terms. Documents were obtained through queries submitted to Google. This corpus contains just over 200 pages. It is representative enough because it came from the actual site, documents deal with 8 topics of interest, and some of these topics may be related to each other by sharing significant terms

A user, who seeks information about lung cancer, could do this through the query:

(cancer OR tumor) AND (lung OR lungs)

After the search is executed, the information displayed is as follows:

```
=== SEARCH WITH XM-TREE BASED SYSTEM===
structure: 8 ints, 47 leaves, 224 data, total 279
revised  : 7 ints (87.50%) 14 leaves (29.79%) 96 data (42.86%)
pruned  : 30              mistakes  : 0
XM-Tree Results: 29    Precision : 1.00    Recall : 1.00

== COMPARATION AGAINST INVERTED INDEX SEARCH ==
                    (173 results)
not retrieved by  XM-Tree       : 144
not retrieved by  inverted index: 0

== COMPARATION AGAINST EXHAUSTIVE SEARCH ==
                    (29 results)
not retrieved by XM-Tree            : 0
not retrieved by  exhaustive search: 0
```

Structure reports the amount of internal nodes, leaf nodes and data of the constructed tree. Data amount coincides (matches) with the total amount of documents indexed, 224 in this case. Next line shows amount and percentage of each type of *revised* nodes in the search XM-Tree. Revision of a node involves computing the *f* norms of XM-Tree. A low percentage of nodes revised indicates a good grouping of similar data in the XM-Tree leaves. Therefore, search algorithm goes to few leaves and prunes many nodes. The amount of pruned nodes corresponds to the amount of subtrees discarded by the RangeSearch algorithm following pruning criteria. This is to say, internal nodes and leaves without relevant data. In this case is quite high, 30 prunings on 55 internal nodes and leaves. This is because the high specificity of the query. The amount of mistakes is zero because the search in the tree properly retrieves all the data that verify the L_∞ similarity criterion. Precision and Recall Indicators of XM-Tree retrieval in their optimal settings reflect the high effectiveness of the search in that index. It obtains all relevant documents and only relevant documents, as it is verified in comparison with the exhaustive search.

Searching emulating an inverted index system gets 173 data corresponding to posting lists of posted terms. This data was subsequently (later) merged and intersected. The amount of XM-Tree hits is in this case 29. This much smaller value verifies the best performance of XM-Tree for this query.

The file with web pages will be re-sorted randomly, to analyze the dependency or not of the XM-Tree performance regard to the order of data insertion. To this, a program built five data sets with pages in different orders. We obtained the same values of Precision and Recall, for each of the data sets. Therefore, it is not important the order of entry documents. Then, we analyze the efficiency of the search in the tree built on each set.

In Figure 1 each line corresponds to one of the queries and shows the percentage of revised data in each set. The data sets are numbered from 1 to 5. As can be seen, every query does not show steep differences between a set and another.

This shows, in terms of index effectiveness, that the tree groups in same way similar pages regardless of the order of the set on which they were built. Hence, similar values of Precision, Recall and revised data, strongly suggest that efficiency and effectiveness of the system is independent of the order in which pages are indexed.

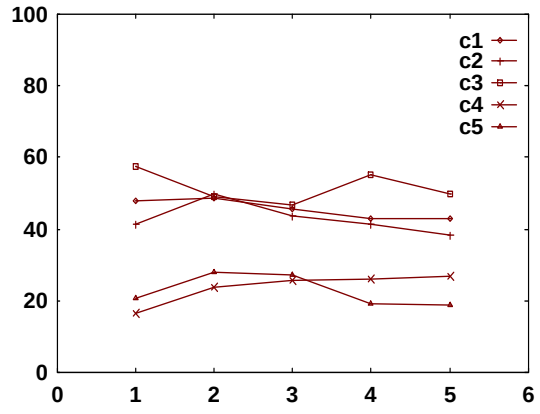

Figure 1. Percentage of revised data for each data set

The second aspect analyzed is the arity *m* of the tree, this is the size of each node. Figure 2 shows percentage of revised data on each query for different arities: 5, 10, 15, 20 and 25.

Figure 3 shows the total amounts of nodes of different trees with the mentioned values of *m*. We can see how as *m* grows, the tree uses less space but more data are revised, because leaves nodes contain a lot of data and the searching algorithm checks them completely.

A loss of efficiency even more important is the slightest ability to group data in a XM-Tree with high arity, because the grouping of similar data in different sub-trees happens after that a node overflows during the insertion of a new datum. This situation does not occur often in a tree with large nodes.

Therefore, the choice of *m* does not affect the quality of results. But, we must choose, in principle, an intermediate value according to the size of the collection (this is, an arity whose tree occupies a reasonable amount of space). With the successive introduction of new data into the collection and their insertion into the tree, the reasonable use of space lows, but the tree does not loss efficiency in query resolution.

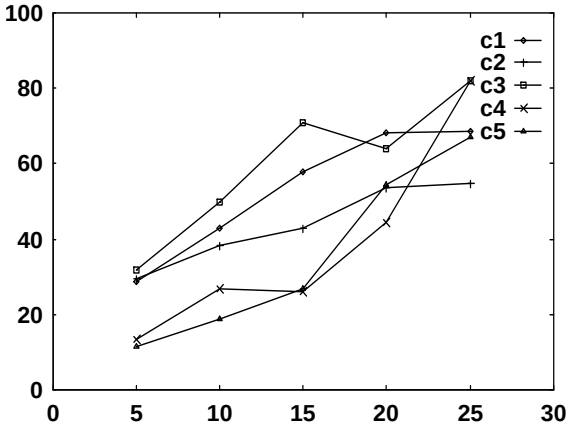

Figure 2. Percentage of revised data for *m*

We preserved the XM-Tree with arity 10 during the next phase of experimentation, because with a lower value the waste of space is very evident, as Figure 3 shows with *m*=5. With higher values the percentage of revised data tends to increase (Figure 2).

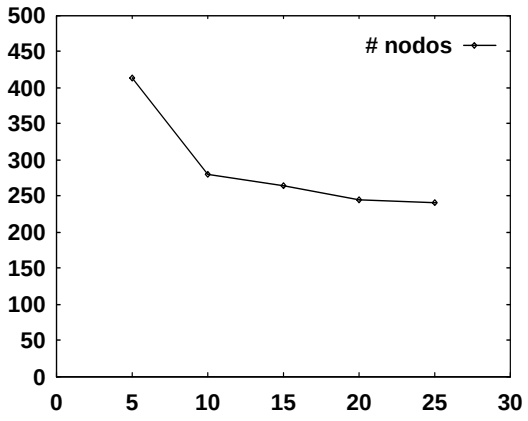

Figure 3. Number of nodes for arity *m*

The last phase of experimentation was to compare the performance of our proposed system and an inverted index

system. We added ten queries to the initial five queries used for the initial experiments. These new queries deal the rest of the main topics in the collection and a number of them are very specific in their topic.
In Figure 4 queries are numbered with x = 1, 2, ..., 15, the values of y show: the percentage of data retrieved by an inverted index system, the percentage of data revised by the XM-Tree at the search, the percentage of retrieved data by the XM-Tree, and the amount of pruned nodes by the XM-Tree. This graphic shows how the amount of revised data by the XM-Tree is proportional to the number of relevant results for each query. This happens because the XM-Tree groups similar documents in its leaves.

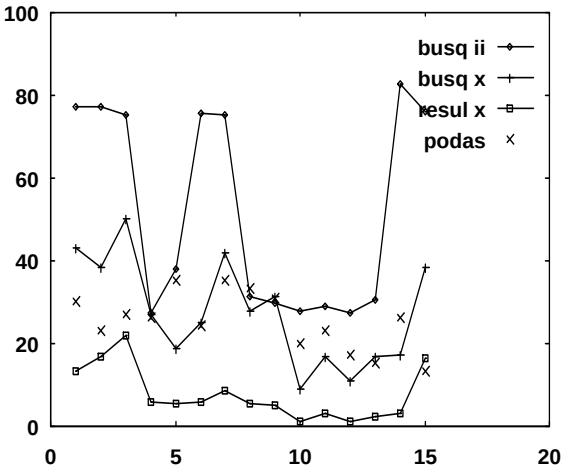

Figure 4. Percentage of revised data, results, and prunings per query

For this reason, the tree revises only the leaves with documents closed to the query. This does not happen in the inverted index system, where the amount of revised data depends directly to the submitted term with more occurrences in the collection, because this term contains large posting lists. Regarding the revision of data collection during a search, our proposed system performs a cost proportional to the amount of relevant documents to the query.

6. CONCLUSIONS

Fast retrieval and quality of results are two needed properties in any information retrieval system. Metric spaces have got indices that allow to retrieval objects which are closed to a given element in a fast and quite accurate way, so they are promising structures in order to build search engines.
This paper proposes the XM-Tree, an index on vector spaces. It is an extension of the M-Tree, and like it, XM-Tree is a paged, balanced and dynamic structure that indexes data in a metric space (including a space vector), resolves range queries, optimizes the execution of the search in a way that reduces the amount of revised data and computed distances, and it is apt for high-dimensional vector spaces.
The extension consists in treating components of vectors separately, in order to adapt the search algorithm of M-Tree to a similarity criterion for Web Information Retrieval. This structure indexes Web documents represented in TF schema uses L_2 as indexing distance and L_∞ as similarity criterion between query and document.
The XM-Tree achieves high performance in terms of quality of results, in a process of Web Information Retrieval, reaching good values of Precision and Recall. Moreover, the efficiency of searches offers significant improvements on vector spaces and inverted indexes. Regarding vector spaces, the XM-Tree groups documents properly allowing go only to the documents closed to the query. Unlike inverted indexes, the XM-Tree revises a fraction of the collection proportional to the set of relevant documents. The experimental results show the realization of the two goals: quality of results and speed in query resolving.
A future extension is to deal with the real deletion of data. Currently, the offline pages still present in the index and list as accessible from a cache, like M-Tree do it.

7. REFERENCES

[1] E. Chávez, G. Navarro, R. Baeza-Yates y J. L. Marroquín. "Searching in Metric Spaces", ACM Computing Surveys, 33(3), 2001; pp 273–321.
[2] R. Baeza-Yates y B. Ribeiro-Neto. (eds.). Modern Information Retrieval. ACM Press, New York, 1999.
[3] P. Ciaccia, M. Patella y P. Zezula. "M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces", Proc. of the 23rd Conference on Very Large Databases (VLDB'97), 1997; pp. 426–435.
[4] C. Traina Jr., A. Traina, B. Seeger y C. Faloutsos. "Slim-trees: High Performance Metric Trees Minimizing Overlap between Nodes", Proc. of 7th International Conference on Extending Database Technology, LNCS, 1777, 2000; pp. 51–68.
[5] X. Zhou, G. Wang, J. Xu Yu y G. Yu. "M^+-tree: A New Dynamical Multidimensional Index for Metric Spaces", Proc. of the 14th Australasian Database Conference, 2003; pp 161 – 168.
[6] X. Zhou, G. Wang, X. Zhou y G. Yu. "BM^+-tree: A Hyperplane-based Index Method for High-Dimensional Metric Spaces", Proc. of 10th International Conference Database Systems for Advanced Applications, LNCS, 3453, 2005; pp 398–409.
[7] M. R. Vieira, C. Traina Jr., F. J. T. Chino y A. J. M. Traina. "DBM-Tree: Trading Height-Balancing for Performance in Metric Access Methods", Journal of the Brazilian Computer Society, v. 11, n. 3, 2006; pp 20.
[8] A. Ocsa y E. Cuadros-Vargas. "DBM*-Tree: An Efficient Metric Acces Method", Proc. of ACM Southeast Regional Conference, 2007; pp 401–406.
[9] P. Ciaccia y M. Patella. "The M^2-tree: Processing Complex Multi-Feature Queries with just one Index", Proceedings of DELOS Workshop: Information Seeking, Searching and Querying in Digital Libraries, 2000.
[10] S. Brin y L. Page. "The Anatomy of a Large-Scale Hypertextual Web Search Engine", Proc of the Seventh International World Wide Web Conference, vol. 30 of Computer Networks and ISDN Systems, 1998, pp. 107–117.
[11] D. J. Delorie. DJGPP. Copyright (c) 2003 URL http://www.delorie.com/djgpp/