Efficiency-Quality Tradeoffs for Vector Score Aggregation

Pavan Kumar C. Singitham
Stanford University
Stanford
USA
pavan@cs.stanford.edu

Mahathi S. Mahabhashyam
Stanford University
Stanford
USA
mmahathi@cs.stanford.edu

Prabhakar Raghavan
Verity Inc.
Sunnyvale
USA
pragh@verity.com

Abstract

Finding the ℓ nearest neighbors to a query in a vector space is an important primitive in text and image retrieval. Here we study an extension of this problem with applications to XML and image retrieval: we have multiple vector spaces, and the query places a weight on each space. Match scores from the spaces are weighted by these weights to determine the overall match between each record and the query; this is a case of *score aggregation*. We study approximation algorithms that use a small fraction of the computation of exhaustive search through all records, while returning nearly the best matches. We focus on the tradeoff between the computation and the quality of the results. We develop two approaches to retrieval from such multiple vector spaces. The first is inspired by resource allocation. The second, inspired by computational geometry, combines the multiple vector spaces together with all possible query weights into a single larger space. While mathematically elegant, this abstraction is intractable for implementation. We therefore devise an approximation of this combined space. Experiments show that all our approaches (to varying extents) enable retrieval quality comparable to exhaustive search, while avoiding its heavy computational cost.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

1 Overview: score aggregation

We have n records $E = \{e_1, e_2, \ldots, e_n\}$ and s sources of evidence. For $1 \leq i \leq s$, we have a *source score* $\sigma_i(e_j)$ from source i for record e_j. Additionally, we have a positive real weight w_i for each of the s sources. For a specified positive integer ℓ, we seek the ℓ records of highest *aggregate score* defined as

$$S(e_j) = \sum_{i=1}^{s} w_i \sigma_i(e_j).$$

In the absence of further structure to exploit, no better algorithm is known than to compute all the $\sigma_i(e_j)$'s and then compute all the $S(e_j)$'s in identifying the top ℓ records. Can we perhaps determine ℓ records almost as good as the ℓ best without such exhaustive search? Note that the ℓ-nearest neighbors problem is a special case of this general setting, in which $s = 1$ and the source score is a geometric proximity measure between a query and the records (represented as points). We focus on an important case of score aggregation, motivated below.

1.1 Motivation

A series of papers motivated by the GARLIC [7] and QBIC [19] systems led to work on *score aggregation* [16, 17]. Recent work on the special case of *rank aggregation* [8, 11, 14, 18] focuses on merging lists of documents ranked by multiple search engines. We detail two motivating applications:

1. In applications like Query By Image Content (QBIC) [19], a user specifies the relative contributions of score components such as color, texture, etc. Each component assigns a score to each record (image) with respect to the query at hand.

2. In semi-structured retrieval for text and XML, it is important to be able to weight the contributions of various elements to an overall score. This can range from simply weighting keywords in text search [15, 27] to weighting fields in a semi-structured document ("retrieve *and rank* books

with `Aho` in the **author** and `algorithm` in the **title**, with the **author** score being twice as important as the **title** score": the notion is that the **author** and **title** fields each contribute a non-negative score that is weighted and summed for the overall score). Already a component of enterprise information retrieval platforms, such functionality becomes even more critical in content-oriented XML retrieval.

1.2 Vector score aggregation

No better algorithm is known for general score aggregation short of an exhaustive search. We focus here on an important case raised by the two examples above. Suppose that record e_j is represented by an s-tuple of vectors $V_{j,i}, 1 \leq i \leq s$, in s vector spaces. For example if each record is a semi-structured document, we would have one vector space for **author**, one for **title**, etc. Each vector space is built from the terms in that field, as in classic information retrieval [30].

Definition 1 *A* composite vector query *is a pair* $\mathbf{Q} = (\mathbf{q}, \mathbf{w})$ *where* $\mathbf{q}$ *is an* s*-tuple of query vectors* $(q_1, \ldots, q_s)$ *in the corresponding vector spaces, while* $\mathbf{w}$ *is an* s*-vector of non-negative real weights* $w_1, \ldots w_s$.

The weight w_i represents the importance assigned by the user to ith field; without loss of generality, we henceforth assume that $\sum_{i=1}^{s} w_i = 1$. We further assume (as is typical in these applications) that the query and record vectors are all normalized within their respective fields, i.e., $||q_i||_2 = ||V_{j,i}||_2 = 1$.

Definition 2 *The* match score *between query* $\mathbf{Q} = (\mathbf{q}, \mathbf{w})$ *and record* $e_j = (V_{j,i})$ *is given by*

$$Match(\mathbf{Q}, e_j) = \sum_{i=1}^{s} w_i (q_i \cdot V_{j,i}), \qquad (1)$$

where $q_i \cdot V_{j,i}$ *represents the dot product (a.k.a. cosine similarity) between the query and record vector* $V_{j,i}$ *in the* i*th field.*

Our problem then becomes: given a composite vector query, can we retrieve the ℓ records of highest match score? In other words, how can we exploit the fact that the s source scores are vector cosine similarities? Note that for $s = 1$, this becomes the traditional ℓ*-nearest-neighbor* problem. Even for this special case of computing the ℓ nearest neighbors in arbitrary dimensions, there appears to be no algorithm that in the worst case avoids exhaustively computing the similarity of the query to every record [1, 10, 23, 25]. This prompts the question: can we find ℓ records that are "almost as good" as the exact ℓ nearest neighbors, while paying significantly less than exhaustive similarity computation? In application settings, an approximation is generally acceptable provided the quality is high enough. For instance, a document or image scoring 0.83 is not likely to be much worse than one scoring (say) 0.87; there is already a (perhaps bigger) approximation in using cosine similarity as a proxy for the user's perception of quality.

We therefore study *efficiency-quality tradeoffs:* suppose that an algorithm $\mathcal{A}$ outputs a candidate set $C = \mathcal{A}_\ell(\mathbf{Q}, E)$ of ℓ records[1]. Can we trade off the computational effort of $\mathcal{A}$ against the quality of C?

To study this question, we must first pinpoint the answers to two questions: (1) how do we quantify the computational effort of $\mathcal{A}$ in a principled manner independent of the scheme $\mathcal{A}$? (2) how do we measure the goodness of a candidate set $C = \mathcal{A}_\ell(\mathbf{Q}, E)$ computed by $\mathcal{A}$? Once we address these questions, we have a basis for comparing various algorithms.

1.3 Metrics

Computational cost: We seek a measure of computational effort that is independent of a particular runtime environment. For any algorithm $\mathcal{A}$, a basic operation is the query-to-record score computation in equation (1) – specifically, this involves s inner product computations. We therefore adopt the number of such query-record score computations by $\mathcal{A}$ as the fundamental measure of work; we denote it by $\mathcal{CC}_\ell(\mathcal{A}, \mathbf{Q}, E)$. We can thus speak of the work done by $\mathcal{A}$ on a query, a query suite, etc. For exhaustive search, $\mathcal{CC}_\ell(Exhaustive, \mathbf{Q}, E)$ is always n. Our interest is in algorithms $\mathcal{A}$ for which $\mathcal{CC}_\ell(\mathcal{A}, \mathbf{Q}, E) \ll n$, while delivering candidate sets of high quality.

Quality of results: To evaluate the performance of an approximate retrieval scheme $\mathcal{A}$ on a given dataset E and query suite $\mathbf{Q}_1, \ldots, \mathbf{Q}_m$, we use a benchmark called the *ground truth*. For each query $\mathbf{Q}$, let the true set of ℓ highest scoring records be $GT_\ell(\mathbf{Q}, E)$. We compare the quality of a candidate set of ℓ records output by an algorithm $\mathcal{A}$ against $GT_\ell(\mathbf{Q}, E)$. To this end, we employ two measures of quality. (For our experiments in Section 4 we use exhaustive search to compute $GT_\ell(\mathbf{Q}, E)$.)

1. The *aggregate goodness* measure

$$AG_\ell(\mathcal{A}, \mathbf{Q}, E) = \sum_{e \in \mathcal{A}_\ell(\mathbf{Q}, E)} Match(\mathbf{Q}, e).$$

Simply put, this is adding up the match scores of the ℓ records returned by $\mathcal{A}$. The idea is that if this net is suitably high, then the user has been given a set of images/documents almost as good as the ground truth. By itself, AG_ℓ does not tell the whole story; for instance, $\mathbf{Q}$ may be a query for which the ground truth does not contain good matches. Rather, we will typically compare

[1] Any algorithm $\mathcal{A}$ in fact implies an ordering of the n records in E with respect to the query $\mathbf{Q}$; thus, $C = \mathcal{A}_\ell(\mathbf{Q}, E)$ consists of the first ℓ in this ordering.

AG with the aggregate goodness of the ground truth, measured by $\sum_{e \in GT_\ell(\mathbf{Q},E)} Match(\mathbf{Q}, e)$; in fact our experiments will compare these quantities averaged over a query ensemble rather than on a single query.

2. The *competitive recall* of the top ℓ results

$$CR_\ell(\mathcal{A}, \mathbf{Q}, E) = |\mathcal{A}_\ell(\mathbf{Q}, E) \cap GT_\ell(\mathbf{Q}, E)|.$$

This computes the fraction of the ground truth included in $\mathcal{A}$'s candidate list of ℓ best records. It is more stringent than aggregate goodness in that it gives no credit for a document that may be almost as good as those in $GT_\ell(\mathbf{Q}, E)$. Note that it hinges on comparison with the ℓ best records for each ℓ, rather on the Boolean notion of relevance commonplace in defining precision and recall in information retrieval. In this sense, our notion of competitive recall is related to the competitive analysis of algorithms [29] and is also related to measures used in [20, 32].

All of the above definitions can be extended to an average over a query suite in the natural way.

2 Summary of contributions

We begin by summarizing related prior work in two broad areas: score aggregation and nearest neighbors. We do this in some depth (Section 2.1.1) for a particular approach to nearest neighbors in vector spaces, that we call *cluster pruning*. We do so because cluster pruning is basic building block for the subsequent development of our approaches.

2.1 Related prior work

A series of papers [14, 16, 17] have looked at the problem of retrieving the ℓ best records from combining source scores. They consider the general (not vector) score aggregation problem and insist on finding the ℓ best results rather than ℓ good results as we do. Their focus is on comparing, for a given instance (records, score function and query) the computational cost of an algorithm in comparison to that of the best algorithm, *on that instance*. This in the worst case could mean a computational cost of n; we instead seek ways of spending far less computation and getting good matches. Rank aggregation – the special case in which each source orders the records without assigning scores – owes its roots to voting theory, but has enjoyed a modern renaissance with the advent of metasearch engines [11, 18].

Nearest neighbor problems in vector spaces are the special case $s = 1$ of vector score aggregation. A series of index structures have been developed for this problem in various settings [2, 3, 21, 24, 26, 34]. These studies use the CPU and disk I/O times during query processing as a measure of speed, in contrast to our higher-level measure of the number of cosine computations. ClusterTree [35] creates an index over the data set that is a hierarchy of clusters and subclusters. The nearest neighbors to a given query are obtained by performing a depth first search in this hierarchy. This approach effectively prunes the search space. They examine the number of such clusters to be probed in order to find all the ℓ nearest neighbors – this can be viewed as one extreme in our tradeoff space (with no approximate near-neighbors). This experimental approach is instructive but may be hard to use directly – in practice we do not have a "stopping condition" that informs us the instant we have found the correct ℓ nearest neighbors. In theoretical work related to approximate nearest neighbors, [23, 25] reduce the problem to point location in equal balls and suggest bucketing and locality sensitive hashing algorithms.

More recently [20] show how simple k-means clustering can do well at approximate nearest neighbor retrieval in multimedia databases. They evaluate quality by metrics that are the complement of competitive recall, and by a matching distance measure. They focus on "progressive processing" of approximate nearest neighbor searching: the user looks at the results for a query, one page at a time. They use approximation techniques with exact nearest-neighbor algorithms to progressively improve results quality as the user keeps looking at more results.

2.1.1 Cluster pruning

We build on a class of schemes for the ℓ-nearest neighbors problem that make use of clustering (the special case of our problem where $s = 1$). The goal is to avoid paying a cost of n cosine similarity computations, while still retrieving ℓ "reasonably near" neighbors for any query. The generic idea is to first cluster the vectors in the dataset E, in the process appointing a *representative* for each of the K clusters [5, 22, 32]. Given a query, we first find the $m \ll K$ centroids nearest to the query and then compute cosine similarities from the query *only* to the records in the clusters represented by these m centroids. All records in all other clusters are ignored. The hope (with no absolute guarantee of course) is that many of the near neighbors are in these m clusters. Thus we get near neighbors while avoiding similarity computations with the majority of the vectors in the dataset.

The clean nature of cluster pruning raises hope that it can be extended to $s > 1$; while this is the idea underlying our approaches, some interesting challenges and design decisions arise.

2.2 Contributions of this paper

- *Concrete, usable metrics for cost-quality tradeoffs that do not demand human relevance judgements as in the TREC evaluations [36].* The idea of quantifying the cost-quality tradeoff for scoring

has not been systematically studied, even for the traditional ℓ-nearest neighbor problem. All our metrics can be applied to the general setting at the beginning of Section 1.

- *Two broad approaches to vector score aggregation:* one inspired by resource allocation (Section 3.1) and the other by ideas from computational geometry [12] (Section 3.2).

- *Experiments with two variants of our scheme based on resource allocation (Section 4), as well as with the scheme inspired by geometry.* We find that all the schemes attain close to the quality of results in the ground truth, at a computational cost dramatically lower than exhaustive search.

- *A comparison of the two families of schemes.* While the geometric indexes are larger than those from resource allocation, they offer better retrieval quality for a given amount of computation on a query.

3 Two approaches

3.1 Resource allocation schemes

The technical development of our schemes inspired by resource allocation is cleaner if we think in terms of a fixed budget B of the computational effort (number of cosine similarities) that we can use to answer a query. We can then ask how well we perform on the quality of retrieved results for the given budget. We begin with the general idea.

Consider again s vector spaces, one for each field. In seeking a candidate set of ℓ records for a composite vector query $\mathbf{Q} = (\mathbf{q}, \mathbf{w})$ we instead retrieve a set C_i of candidate records from the ith field, for each $i \in [1, s]$. Finally, we return the ℓ best matches from the records in $\cup_{i=1}^{s} C_i$.

These retrievals C_i for each i use the cluster pruning scheme in Section 2.1.1; the precise implementation details and parameter choices are deferred for now. Essentially, we first retrieve nearly best matches from each field, then pick the ℓ best matches from among these candidates. An important question arises: given our budget of B for computational effort, how do we invest this budget across the s vector spaces? This is a resource allocation problem and we study two natural schemes for this investment. For example, consider a simple allocation between two fields **author** and **title**. Suppose that a query places a high weight on the **author** field and relatively little weight on the **title** field. We could on the one hand spread our budget equally in the **author** and **title** vector spaces. On the other hand, we could invest more of our budget into retrieving candidates from the **author** space rather from the **title** space, as this might give us better score-aggregated quality.

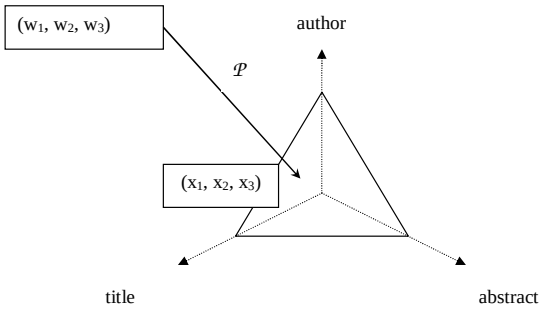

Figure 1: Mapping from query weights to points in the simplex: resource allocation.

Note that at query time we could make use of the weights in $\mathbf{w}$ to determine this allocation. Interesting questions arise: should we? If so, how do the weights govern the allocation? This question may be viewed at a slightly higher level (for conceptual development only; eventually we will use the number of cosine similarities as the measure of computational effort). Rather than a budget B of cosine similarities, imagine a budget P of the number of *probes*: a probe is a decision to evaluate the query against all the records in any single cluster in any of the s spaces. This view reflects the working of our algorithms built on cluster pruning: the query is always evaluated against all vectors in a cluster, or none.

A weight vector $\mathbf{w}$ in a query can naturally be viewed as a point on the s-dimensional simplex $\sum_{i=1}^{s} w_i = 1$. Further, any point $\mathbf{x} = (x_1, \ldots, x_i, \ldots, x_s)$ on this simplex represents an allocation as follows: given a budget of P probes, we dispatch $x_i P$ probes into field i. Figure 1 shows this idea for $s = 3$ where the fields are **author**, **title** and **abstract**.

Definition 3 *A* probe resource allocation *is a mapping $\mathcal{P}$ from the simplex onto itself, $\mathcal{P} : \mathbf{w} \rightarrow \mathbf{x}$.*

What properties should hold for an allocation function $\mathcal{P}$? We propose these below; of these the first two should clearly hold for all $\mathcal{P}$, while the remainder are plausible but (as we detail in Section 5) may not hold in all situations.

1. *$\mathcal{P}$ should map each vertex of the simplex into itself.* This simply says that a query that places all its weight into one field demands that all the probes go into that vector space index.

2. *$\mathcal{P}$ should map each edge (in general, lower-dimensional simplex) of the simplex into itself.* This says that if a field gets no weight in the query,

Algorithm 1 *Uniform.*

```
number of cluster probes available = P; Query Q;
SearchSet = ∅;
for i ← 1, 2, ..., s do
    NSet= set of P/s nearest clusters to Q taken from field i;
    SearchSet = SearchSet union records in NSet;
end for
for all record ∈ SearchSet do
    compute Match(Q,record)
end for
Rank SearchSet based on Match
```

Algorithm 2 *Transparent.*

```
number of cluster probes available = P; Query Q;
SearchSet = ∅;
for i ← 1, 2, ..., s do
    NSet= set of w_i*P nearest neighbor clusters to Q from field i;
    SearchSet = SearchSet union records in NSet;
end for
for all record ∈SearchSet do
    compute Match(Q,record)
end for
Rank SearchSet based on Match
```

its vector space should not be probed. This is a more stringent requirement than (1) above.

3. *$\mathcal{P}$ should map the center of the simplex into itself.* In other words, if the query calls for a uniform weighting on the fields, the investment should be uniform; this should recursively hold for any lower-dimensional simplex. In Figure 1, the recursive requirement means that the mid-point of each side of the triangle maps into itself.

4. *$\mathcal{P}$ should act as the identity mapping on each edge (lower-dimensional simplex) of the simplex.* This demands that if any field gets zero weight, the investment in the other fields is *directly proportional* to their weights. In Figure 1 this implies the identity mapping for the perimeter of the triangle.

The third and fourth properties raise the question: can $\mathcal{P}$ be the identity mapping itself? This is the second of our two allocations, detailed below in Section 3.1.2.

3.1.1 Uniform allocation

In this scheme, we allocate the budget of cluster probes uniformly across the vector spaces. Thus the query weighting **w** is ignored for probe allocation purposes (but of course remains in use for the score computations and quality measures). *Uniform* is described in Algorithm 1.

3.1.2 Transparent allocation

Here $\mathcal{P}$ is the identity mapping; thus each field receives an allocation of probes in proportion to the query weight for that field. *Transparent* is described in Algorithm 2.

3.2 Cell decomposition indexes

An alternate approach to having separate vector spaces for each field is to combine them all into a single gigantic vector space. This is inspired by ideas from combinatorial geometry which we now review; these are elegant but not pragmatic at the dimensionality we are discussing. Accordingly we will first develop the approach, then coarsen it to make it practical.

Consider first the standard ℓ-nearest neighbors problem, i.e., $s = 1$. The *Voronoi decomposition* [12] partitions space into n polyhedral cells, one for each record e_i. The crucial property: for any query point within e_i's cell, the nearest neighbor is e_i.[2] Given a query, the nearest-neighbor problem reduces to locating which cell the query lies in. Tight bounds exist on the total number of facets in Voronoi decompositions as a function of n and the number of dimensions d; these bounds are exponential in d. Consequently, the computational costs of building the decomposition and for point location are high when d is large; but for $d \leq 3$ this approach leads to pragmatic nearest neighbor retrieval.

The notion of a Voronoi decomposition has been generalized [12] for ℓ-nearest neighbors. Instead of one cell per record e_i, we now have one cell for every subset of ℓ records that is a valid answer to *some* query. The cell decomposition now has the following property: the same set of ℓ records constitute the ℓ nearest neighbors for any query point within a given cell. Thus identifying the ℓ nearest neighbors again reduces to identifying the cell containing the query. Here too the cells are known to be polyhedra (for cosine similarities between unit vectors) and bounds are known for the facet complexities. Despite its impracticality in high dimensions, we nevertheless pursue this view a little further, as it leads to our eventual index structure.

Let us extend the above notions to composite vector retrieval: denote by $D_1, \ldots, D_s$ the s vector spaces and $d_1, \ldots, d_s$ the corresponding dimensionalities. Note additionally that the set of *all possible* query weightings **w** can be viewed as a vector space W in s dimensions (in fact since $\sum_{i=1}^{s} w_i = 1$, a simplex). Consider a new vector space $U = W \cup_{i=1}^{s} D_i$, having $u = s + \sum_{i=1}^{s} d_i$ dimensions. Any query $\mathbf{Q} = (\mathbf{q}, \mathbf{w})$ can be represented as a single point in U; note however that a record is *not* a single point in U. Nevertheless, U can still be partitioned into cells such that for any query (point in U), the set of ℓ nearest records is invariant. In this cell structure, it suffices to locate the

[2] In fact, the shapes of the cells depend on the distance metric; for cosine similarities between unit vectors, we have unbounded polyhedral cones whose apices are at the origin.

query point then read off the ℓ nearest neighbors for any query in that cell. Besides the immense number of cell boundaries due to the high dimensionality, there is an added difficulty here: the cell boundaries are no longer polyhedral, but rather described by (nonlinear) algebraic functions. Point location thus becomes highly non-trivial.

We give two generic ideas to overcome these difficulties, leading to experimentation with a very basic implementation of these ideas in Section 5.4.

1. We can group together cells in the decomposition that are "close together", coarsening the decomposition into a small number of *coarse cells* with similar (rather than the same) answers. In the process, we may project U down to a low-dimensional space.

2. We can approximate the cell boundaries by linear functions.

For any such coarse approximation U' of U, we now have to address (1) point location in a coarse cell of U'; (2) for each coarse cell, an index tuned to efficiently retrieve ℓ high-quality records for that cell. This is necessary since there is no longer a unique set of ℓ answers within a coarse cells.

Example 1 *We begin with an extremely simple manifestation of the above ideas. Suppose we have three vector spaces* **author**, **title** *and* **abstract** *as in Figure 1. Each query has three weights w_{author}, w_{title} and $w_{abstract}$, together with corresponding query vectors q_{author}, q_{title} and $q_{abstract}$. For any query in which $w_{author} \geq 0.34$, we simply find the ℓ nearest neighbors to q_{author} in the* **author** *vector space alone, ignoring the other fields. Similar rules can be invoked for the* **title** *and* **abstract** *fields.*

The intuition of this simplistic scheme: if the query places the greatest weight in a field, we run a vector-space query for that field alone and ignore the rest of the query. Notice that this can be viewed as a projection of the huge vector space developed above down to a simplex in three dimensions, a coarsening of this simplex into three regions, and finally an efficient (if not perhaps high-quality) retrieval scheme for all queries falling in each one of these regions. Just as we did for allocation maps $\mathcal{P}$ in Section 3.1, we can enumerate basic symmetry requirements for any version of this scheme; we omit these for brevity here. In Section 5.4 we experiment with a slightly more sophisticated version of this scheme. The generic cell decomposition retrieval algorithm is given in Algorithm 3 *CellDec*.

3.3 Comparing the schemes

In this section we compare the resource usages of the resource allocation and cell decomposition schemes. For the allocation schemes, we would need to maintain

Algorithm 3 *CellDec.*

```
1: number of cluster probes available = P;
2: Query Q = (q,w);
3: Identify the cell decomposition index of the coarse
   cell i based on the query template.
4: NSet = set of P nearest clusters to Q from index
   of coarse cell i;
5: SearchSet = Union of records in NSet;
6: for all record ∈SearchSet do
7:    compute Match(Q,record)
8: end for
9: Rank SearchSet based on Match
```

s separate indexes, one for each of the s vector spaces. For *CellDec* the number of indexes maintained, r, is the number of *coarse cells* used in the decomposition. In our running examples with three fields (**author, title** and **abstract**), we use 3 indexes for the allocation schemes but r indexes for *CellDec*. In Example 1, $r = 3$; in the version we experiment with in Section 5.4, $r = 4$. The index size is arguably larger for *CellDec*, since we are looking at a combined vector space representing all features spaces and their dimensions. On the other hand, there is a trade-off involved in the computational cost at query-time; we study this now. The computational cost(number of cosine similarity computations) stems from two sources in all schemes derived from cluster pruning (including all ours): cosine computations for

- (query,cluster centroid) pairs and
- (query,records in the set `SearchSet` from the algorithms above). This measure is an invariant in the number of scalar multiplications, across all the three algorithms above, because the $Match$ computation is over the entire record irrespective of the higher level indexing scheme used.

For the allocation schemes, the total computation cost is

$$s \cdot K + |\texttt{SearchSet}| \tag{2}$$

where K is the number of clusters in each of the s fields. For *CellDec*, with each index having K clusters each as well, the cost is

$$K + |\texttt{SearchSet}|$$

since we are exploring only one *coarse cell* index for a query. In both cases, if n is the total number of records in the data set,

$$E[|\texttt{SearchSet}|] = O(Pn/K)$$

where $E[]$ denotes the expectation of a random variable. Thus with *CellDec* we gain an advantage of nearly $(s-1)K$ cosine computations at query time, by investing all the P probes into one *coarse cell* index. A point to keep in mind though is that the advantage is not exactly $(s-1)K$, because of the potential difference in centroid lengths of the two schemes.

4 Experimental setup

We now describe the data used in our experiments, followed by the query suite. In Section 5 we describe our findings on the computation-quality tradeoff.

4.1 Data set and preparation

We perform our experiments on a data set obtained from crawling citeseer [37]. This data consists of 480,000 documents; for each document, we have three fields – **author, title** and **abstract**. This data is processed by stemming and stop-word elimination (standard data preparation steps in information retrieval [30]) and inserted into three *base tables* in a mySQL database. For each of the fields, the frequency of each term (tf) is computed and normalized; thus, within each field for each document, the squares of the frequencies of various terms add up to one. At this point we have three vectors for each document, one for each field.

Thus if a vector has m features with term frequencies $\{tf_1, ..., tf_m\}$, the weight w_i of the ith term is

$$tf_i / \sqrt{(\sum_{j=1}^{m} tf_j^2)}.$$

4.2 Query suite

Our query suite consists of two sets each having 250 *query prototypes*, each of which is a triplet of vectors corresponding to an instance of **q** in Section 1.2. The first, Set A, is meant to model typical user queries from researchers searching a corpus such as citeseer using composite queries on the three fields. Set B is meant to explore the tradeoffs by systematically neutralizing certain inherent asymmetries in three fields with rather different term distributions (e.g., the **author** field in most documents has fewer than three terms (author names); but few abstract fields have fewer than 30 distinct terms). We motivate Set B further in Section 5.

4.2.1 Query prototypes

For Set A we pick the 250 most popular co-author pairs. From the pool of titles and abstracts of documents authored by each pair, we randomly select words from the 100 most frequent words. This gives us queries of the form ($author_1$, $author_2$, $titleword_1$, $titleword_2$, $abstractword_1$, $abstractword_2$). Thus our query prototypes will not pair (say) authors `Garcia-Molina` (a database researcher) and `Micali` (a cryptographer); because they have not co-authored a paper, it is unlikely that a user is searching for documents co-authored by the pair. Extending the same principle to conditioning the generation of $titleword_1$, $titleword_2$, $abstractword_1$ and $abstractword_2$, we ensure that the query prototypes of Set A are likely to correspond to documents that a user might actually search for. This also ensures that there are likely to be at least some documents in the corpus that are high-quality matches for each query.

For Set B, we use a set of 250 randomly generated queries on a new *synthetic* data set. This new data set has three fields f_1, f_2 and f_3 that are all generated from the **title** field of our original data set. Each *synthetic* document is composed of 3 random *original* document titles, each title forming a field f_i. Given $title_i, i \in 1, 2, ..., n$ of documents $OriginalDoc_i$ in the original collection, the documents $SyntheticDoc_j, j \in 1, 2, ..., n/3$ in the synthetic data set are

$$f_1 = title_j, f_2 = title_{n/3+j}, f_3 = title_{2n/3+j}$$

In this data set the document vector lengths in the three field spaces become comparable. Each query in Set B consists of two terms from each field, generated uniformly at random.

T#	w_{author}	w_{title}	w_{abstract}
1	0.33	0.33	0.34
2	0.4	0.4	0.2
3	0.4	0.2	0.4
4	0.2	0.4	0.4
5	0.6	0.2	0.2
6	0.2	0.6	0.2
7	0.2	0.2	0.6

Table 1: Weight templates.

4.2.2 Weight templates

For each query prototype we apply seven weight templates, each a triplet of weights. The weights in a template sum to one and model skewed user weighting. The templates are given in the Table 1.

Note that templates 2-4 are rotations (around the fields) of each other; likewise for templates 5-7. The first template is meant to model an unbiased query (the user does not emphasize any field); note that for such queries an alternative approach would be to treat the entire document (with all its fields) as one "bag of words" (a single vector) and treat the user query terms also as a single vector. Templates 2-4 model situations where the user emphasizes two fields but is less certain or demanding about the third. Similarly, templates 5-7 model situations where the user emphasizes a single field at the expense of the other two. These broad situations clearly span the gamut of symmetric user needs. The rotations are meant to elicit the effects of asymmetries between the three fields. Templates 5, 6 and 7 are especially useful to study: they can be viewed as a "basis" using which an arbitrary **w** can be expressed as a linear combination; thus results on allocation on these templates can be combined to devise allocations for arbitrary weight vectors.

n= 50K	128	256	512
AG	91.54	93.57	91.80
CR	68.37	71.47	67.09

Table 2: Performance for different values of K for collection size 50,000.

n=100K	128	256	512
AG	87.57	92.53	89.80
CR	64.97	66.82	65.27

Table 3: Performance for different values of K for collection size 100,000.

5 Results and analysis

We implement the traditional K-means algorithm in clustering each of the 3 fields. To represent a centroid of the cluster, we use the mean of the document vectors within a cluster. Each cluster centroid is implemented as a hashtable of terms and the term weights, so that the lookup is much faster while performing a similarity computation between the query and the centroid. In order to nullify the difference in size of the index between the cell decomposition and allocation schemes we do the following:

- Use the same number of clusters K for both kinds of indexes.
- Store only the top 1000 highest weight terms of the mean of all document vectors, in the centroid.

This ensures that centroid lengths and index sizes for both the schemes are the same and we can invoke Section 3.3 to determine $\mathcal{CC}_\ell(\mathcal{A}, \mathbf{Q}, E)$ and $\mathcal{CC}_\ell(\mathit{CellDec}, \mathbf{Q}, E)$; here $\mathcal{A}$ represents either allocation algorithm.

5.1 Choosing the right value of K

Theoretically, the optimal value of K can be estimated as follows. From Equation 2 the computational effort involved for one cluster probe $\mathcal{CC}_\ell(\mathit{Uniform}, \mathbf{Q}, E)$ with 3 vector spaces, is given by $3K + n/K$. This is minimized when $K = \sqrt{n/3}$. For our corpus size of 480K documents, the value of $\sqrt{(n/3)}$ is 400. To further validate this estimate for K, we conduct some experiments. For this, we measure the performance of *Uniform* against the ground truth for different values of K. We experiment with subsets of our document set with 50,000 and 100,000 documents, and cluster them using various values of K. In each case we fix the computational cost at roughly 2500 (there is some variation because when we decide to probe a set of clusters, their sizes may not add up to exactly 2500). The results (for both metrics) are shown in Tables 2 and 3.

We see that the quality peaks around $K = 256$ for both the sample corpora. For $n = 50,000$, the value of $\sqrt{n/3}$ is 128 (approximately) and for $n = 100,000$, this value is 182.

For the collection size $n = 480,000$, we perform the experiments with different values of K.The results are shown in table 4. We choose $K = 450$ as the number of clusters for further experiments on the full data set.

n=480K	300	350	400	450	500
AG	96.69	96.85	97.28	97.38	95.54
CR	80.55	80.88	83.67	83.98	75.12

Table 4: Performance for different values of K for collection size 480,000.

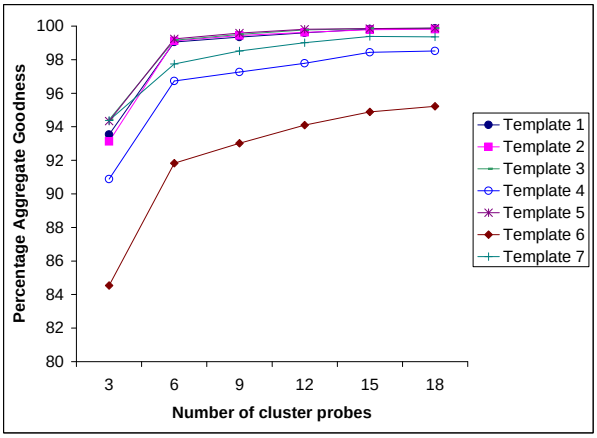

Figure 2: Uniform vs. the ground truth – aggregate goodness.

5.2 Uniform vs. the ground truth

We explore the performance of *Uniform* in further detail. Figures 2 and 3 show its performance for each weight template against the ground truth, for the queries in Set A. The figures illustrate the fundamental tradeoff between computational effort and quality: at low effort the quality (by either measure) is quite modest. By the point where we invest three probes in each field index, we begin to see a significant (but tailing off) improvement. Other key conclusions:

- Cluster pruning even with *Uniform* performs very well in returning high-quality results with a minuscule fraction of the clusters probed (3 out of 450, which means our computational effort is only 0.67% of exhaustive search).
- The y-axis in Figure 2 is the percentage of the aggregate goodness of the ground truth, averaged over the queries. Note that unlike the (more stringent) competitive recall measure, we quickly get close to 100% by this metric. Thus users get documents essentially as good as (if not the same as) the ground truth.

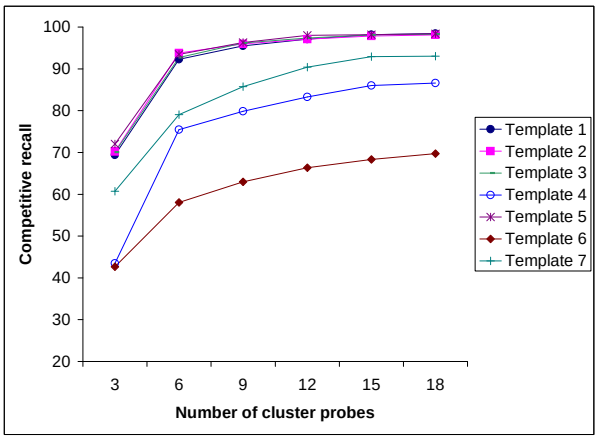

Figure 3: Uniform vs. the ground truth – competitive recall.

We observe that for templates with high weight on the **author** field (Templates 2 and 5), the retrieval quality is much higher than the other templates.

This happens because of two biases:

- The asymmetry of the fields: Each document is represented by fewer nonzero vector components in the **author** field than (for instance) the **abstract** field. Consequently, any match on authors tends to dominate the similarity score more than a similar match on abstracts.
- The query generation scheme for Set A is conditioned by an author pair chosen from the **author** field. The significance is not that our query generation is skewed or misleading. Rather, an application using resource allocation should bias the mapping $\mathcal{P}$ towards the dominant mode by which users think of query tasks (e.g., if they begin by thinking of titles as the primary driver of their queries, $\mathcal{P}$ should invest disproportionately additional work in the **title** index).

5.3 Uniform vs. Transparent

Transparent gives only marginal improvements over *Uniform* on Set A. This stems from our mode of generation of the queries in Set A; so we studied Set B instead to see if a different class of queries would highlight the differences between *Uniform* and *Transparent*. The results are shown in Figures 4 and 5. For these comparisons we show the number of cluster probes P invested on the x-axis (since the computational costs of both the allocation schemes are linear in and proportional to the number of probes invested). We observe that *Transparent* performs consistently better than *Uniform*, for all the templates. In particular, it is interesting to note that it beats *Uniform* especially on highly skewed templates. This suggests that when user needs come from a more homogeneous setting such as Set B, non-uniform allocation

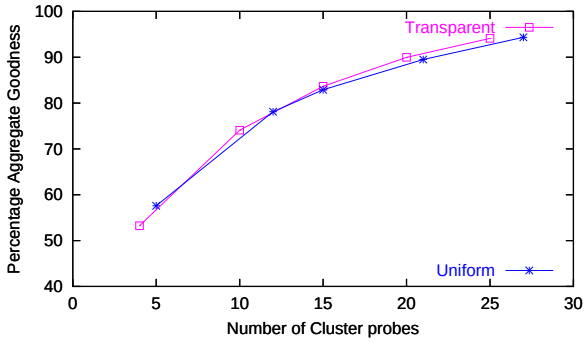

Figure 4: Transparent vs. Uniform - Aggregate Goodness.

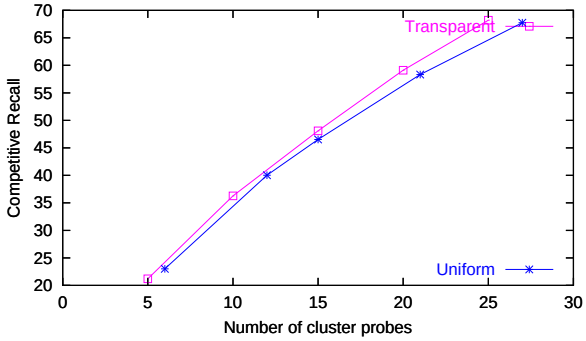

Figure 5: Transparent vs Uniform - Competitive recall.

makes a difference. An interesting area that now opens up: can more sophisticated allocation than *Transparent* make a bigger difference? How do we design the optimal policy $\mathcal{P}$?

5.4 Cell decomposition indexes

We now describe experiments with a slightly more sophisticated cell-decomposition than the simple scheme of Example 1 in Section 3.2. Consider the unit simplex of query weights for each of the three fields, Figure 6. This simplex is partitioned into four cells labeled $1, 2, 3$ and 4. Each cell corresponds to a range of weights that a query can take. We maintain one optimized index for each cell; whenever the weights in a query fall into cell $i, 1 \leq i \leq 4$, we use index i. Recall Table 1 listing the weight templates for our experiments; we thus note that Templates 1–4 fall in region 2, with templates 5, 6 and 7 falling respectively in regions 2, 3 and 4.

Next, we describe the index for each region:

- Region 1: For $1 \leq j \leq n$ and $1 \leq i \leq 3$, let $V_{j,i}$ and denote the vector for record j in field i. For each record j we compute a composite vector

$$V_j = \sum_{i=1}^{3} V_{j,i}.$$

We now build an index based on cluster pruning on the *single* vector space spanned by the

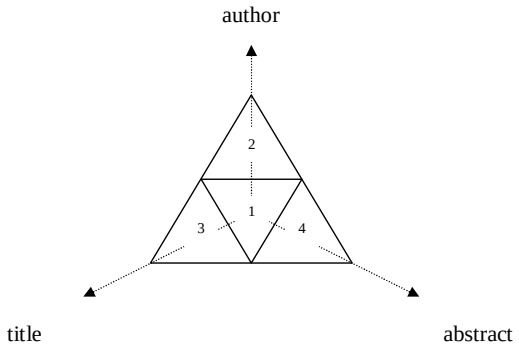

Figure 6: Regions of the triangular simplex covered by each index.

V_j's. Intuitively: in region 1 the query weights are "roughly the same" so we simply treat all the contents of a record – authors, title, abstract – as one bag of words and use cluster pruning on the resulting vectors.

- The index for region 2 is created by a linear combination of the author vector and the vectors from the other two fields – titles and abstracts – the latter each multiplied by a *squeeze factor*, θ. Thus

$$V_j = V_1 + \theta * V_2 + \theta * V_3.$$

The indices for 3 and 4 are also created similarly by squeezing a different pair of axes. Intuitively, we are attenuating the vectors from fields that are de-emphasized in the queries using the particular region.

Note that these indexes are created up front; when a query specifies a particular weight vector $\mathbf{w}$, it is sent to the index that is likely to yield the best *quality* results for that weight vector.

While creating the clusters for cluster pruning, we do K-means clustering of the vectors thus obtained, just as in our earlier schemes. The only difference comes in the computation of the centroid for each cluster. While calculating the mean of the documents within a cluster, we do an L_2-normalization within the terms of each field in a document, before calculating the centroid. This ensures that fields with very few dimensions are not under-represented.

To estimate a good value for the squeeze factor θ, we use a sampled subset of the documents containing 10000 random documents and values of $\theta \in [0.1, 1]$. The results are shown in Figure 7. We observe that for $\theta = 0.5$, queries from all the three templates do the best. Hence we now choose this as our squeeze factor to compare the performance of the cell decomposition scheme with respect to both the uniform and transparent allocation schemes, using the same data set of

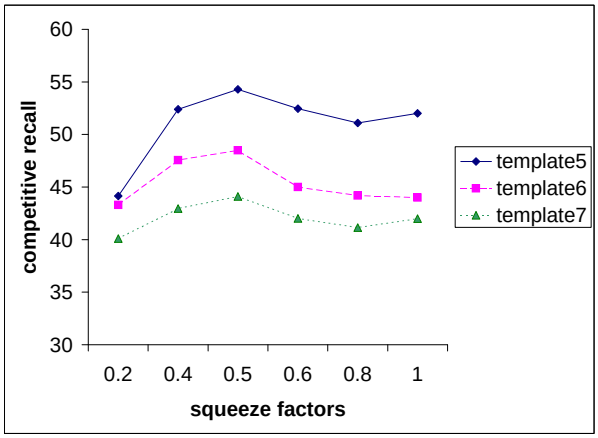

Figure 7: Performance for different squeeze factors.

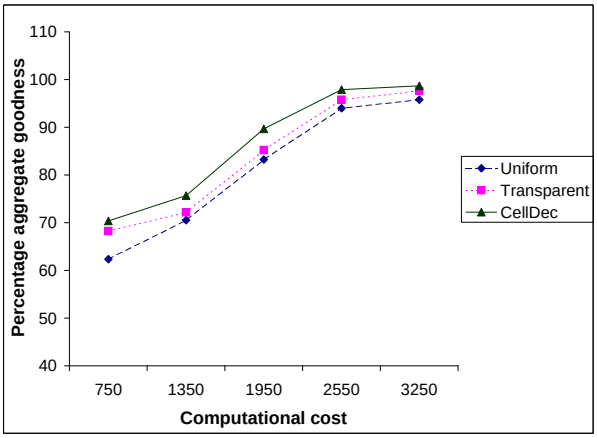

Figure 8: Aggregate goodness vs. Computational cost.

10000 documents. The results are shown in Figures 8 and 9.

We observe that even our simple cell decomposition scheme consistently outperforms both *Uniform* and *Transparent*, whether for a fixed cost or for a fixed quality.

6 Conclusions and further work

Our work (particularly with the aggregate goodness measure) suggests that we can find high quality results for vector score aggregation at a small fraction of the computation of exhaustive search. Our experiments raise the pursuit of more sophisticated allocation schemes. This becomes especially intriguing with recursive cluster pruning schemes, where the allocation at higher levels can depend on what is deeper in each sub-tree. The second area for further work is on more sophisticated cell decomposition schemes: given an application, how do we determine the best cell decomposition scheme based on system parameters? How (for either class of schemes) should the algorithm parameters be data-dependent? Empirically studying cost-quality tradeoffs in more general settings [13, 6]

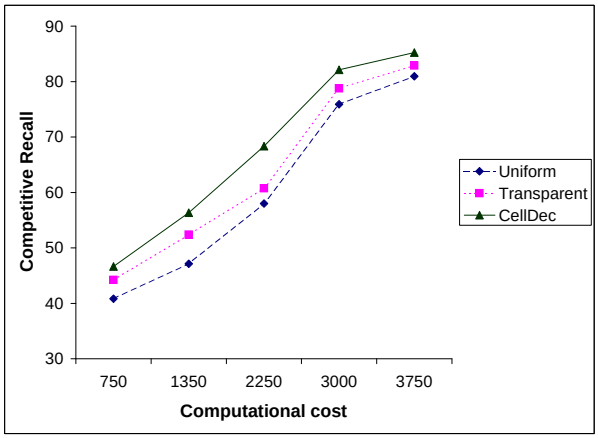

Figure 9: Competitive recall vs. Computational cost.

is an exciting direction.

References

[1] P.K. Agarwal, J. Erickson. Geometric Range Searching and Its Relatives. In *CRC Handbook of Computational Geometry,* 1997.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. *Proceedings of ACM SIGMOD,* 322–331, 1990.

[3] S. Berchtold, D. A. Keim, and H.-P.Kriegel. The x-tree: An index structure for high-dimensional data. *Proc. of the 22th VLDB Conference,* 1996.

[4] M. W. Berry, S. Dumais, G. W. O'Brien. Using Linear Algebra for Intelligent Information Retrieval. *SIAM Review* 37:4 (1995).

[5] S. Bhatia, J. Deogun. Cluster characterization in Information retrieval. *ACM-SAC 1993 Indiana USA, 721-727.*

[6] M. Charikar, R. Fagin, V. Guruswami, J. Kleinberg, P. Raghavan and A. Sahai. Query strategies for priced information. *Journal of Computer and System Sciences* **64***(4):785-819,* 2002.

[7] W. Cody, L. Haas, W. Niblack, M. Arya, M. Carey, M. Flickner, D. Lee, D. Petkovic, P. Schwarz, J. Thomas, M. Tork Roth, J. Williams, R. Fagin and E. Wimmers. Querying multimedia data from multiple repositories by content: the Garlic project. *IFIP 2.6 3rd Working Conference on Visual Database Systems (VDB-3)*, 1995.

[8] M.-J. Condorcet. Essai sur l'application de l'analyse a la probabilite des decisions rendues a la pluralite des voix, 1785.

[9] T. Cover, P. Hart. Nearest Neighbor pattern classification. *IEEE Transactions on Information Theory*, 13 (1967), 21–27.

[10] D. Dobkin, R. Lipton. Multidimensional Search Problems. *SIAM Journal of Computing*, 5 (1976), 181–186.

[11] C. Dwork, R. Kumar, M. Naor and D. Sivakumar. Rank aggregation methods for the web. *Proceedings of WWW10*, 2001.

[12] H. Edelsbrunner. Algorithms in Combinatorial Geometry. Springer-Verlag, 1987.

[13] O. Etzioni, S. Hanks, T. Jiang, R.M. Karp, O. Madani and O. Waarts. Efficient Information Gathering on the Internet, *37th Annual Symposium on Foundations of Computer Science* , 1996.

[14] R. Fagin. Combining Fuzzy information from multiple systems. *Journal of Computer and System Sciences,* **58**(1):83–99, 1999.

[15] R. Fagin and Y. Maarek. Allowing users to weight search terms, *RIAO (Recherche d'Informations Assistee par Ordinateur)*, 682–700 (2000).

[16] R. Fagin and E. Wimmers. A formula for incorporating weights into scoring rules. *Theoretical Computer Science* **239**, 2000.

[17] R. Fagin, A. Lotem and M. Naor. Optimal aggregation algorithms for middleware. *J. Computer and System Sciences* **66**, 2003.

[18] R. Fagin, R. Kumar, D. Sivakumar. Efficient similarity search and classification via rank aggregation *Proceedings of ACM SIGMOD*, 2003.

[19] C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic and R. Barber. Efficient and Effective Querying by Image Content. *Journal of Intelligent Information Systems,* 1994.

[20] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal and A.E. Abbadi. Approximate Nearest Neighbor Searching in Multimedia Databases. *Technical Report TRCS00-24,* Comp. Sci. Dept., UC Santa Barbara, 2000.

[21] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Proceedings of ACM SIGMOD,* 47–57, 1984.

[22] J. Hafner, N. Megiddo and E. Upfal. US Patent 5848404: Fast query search in large dimension database, 1998.

[23] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In Proc. of 30th STOC, 604–613, 1998.

[24] N. Katayama and S. Satoh. The SR-tree: An Index Structure for High-Dimensional N earest Neighbor Queries. *Proceedings of ACM SIGMOD*, 1997.

[25] J. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. *Proc. 29th ACM Symposium on Theory of Computing,* 1997.

[26] K.-I. Lin, H. V. Jagadish and C. Faloutsos. The TV-tree: An Index Structure for High Dimensional Data. *VLDB Journal,* **3**(4):517–542, 1992.

[27] X. Long and T. Suel. Optimized Query Execution in Large Search Engines with Global Page Ordering. *Proceedings of VLDB*, 2003.

[28] D.G. Luenberger. Investment Science. Oxford Press, 1997.

[29] On-line Problems. *Journal of Algorithms*, 11:208-230, 1990.

[30] G. Salton. The SMART Retrieval System – Experiments in automatic document processing. Prentice Hall Inc., Englewood Cliffs, 1971.

[31] T. Sellis, N. Roussopoulos and C. Faloutsos. The R+-Tree: A Dynamic Index For Multi-Dimensional Objects. *VLDB Journal*, 1987.

[32] S. Sitarama, U. Mahadevan, M. Abrol. Efficient cluster representation in similar document search. *Proceedings of WWW conference,* 2004.

[33] I.H. Witten, A. Moffat, T.C. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, 1994.

[34] D.A. White and R. Jain. Similarity Indexing with the SS-tree. *In Proceeding s of the 12th Intl. Conf. on Data Engineering*, 1996.

[35] D. Yu, A. Zhang. ClusterTree: Integration of Cluster Representation and Nearest Neighbor Search for Large Datasets with High Dimensionality. *IEEE Internati onal Conference on Multimedia and Expo*, 2000

[36] http://trec.nist.gov/ : Text Retrieval Conference series.

[37] http://citeseer.nj.nec.com : Citeseer Scientific Digital Library.