Appears in *Proc. of the 16th Int'l Conf. on World Wide Web*, Banff, Alberta, Canada, 131-140, 2007.
(Revised May 21, 2007. This version contains some minor clarifications not present in the original.)

Scaling Up All Pairs Similarity Search

Roberto J. Bayardo
Google, Inc.
bayardo@alum.mit.edu

Yiming Ma*
U. California, Irvine
maym@ics.uci.edu

Ramakrishnan Srikant
Google, Inc.
srikant@google.com

ABSTRACT

Given a large collection of sparse vector data in a high dimensional space, we investigate the problem of finding all pairs of vectors whose similarity score (as determined by a function such as cosine distance) is above a given threshold. We propose a simple algorithm based on novel indexing and optimization strategies that solves this problem without relying on approximation methods or extensive parameter tuning. We show the approach efficiently handles a variety of datasets across a wide setting of similarity thresholds, with large speedups over previous state-of-the-art approaches.

Categories: H.3.3 [**Information Search and Retrieval**]: *Search Process, Clustering*
General terms: Algorithms, Performance
Keywords: similarity search, similarity join, data mining

1. INTRODUCTION

Many real-world applications require solving a similarity search problem where one is interested in all pairs of objects whose similarity is above a specified threshold. Consider for example:

- **Query refinement for web search:** Search engines often suggest alternate query formulations. One approach to generating such suggestions is to find all pairs of similar queries based on the similarity of the search results for those queries [19]. Since the goal is to offer only high quality suggestions, we only need to find pairs of queries whose similarity score is above a threshold.
- **Collaborative filtering:** Collaborative filtering algorithms make recommendations by determining which users have similar tastes. Thus the algorithms need to compute pairs of similar users whose similarity is above some threshold.
- **Near duplicate document detection & elimination:** Especially in the area of document indexing, it is important to detect and purge documents that are equivalent. In many cases, the presence of trivial modifications make such detection difficult, since a simple equality test no longer suffices. Near duplicate detection is made possible through similarity search with a very high similarity threshold.
- **Coalition detection:** Recent work has applied algorithms for finding all similar pairs within an application for identifying coalitions of click fraudsters [13].

While these applications are not all new, the scale of the problem has increased dramatically due to the web. The number of distinct search queries issued over a single week to any large search engine is in the tens of millions. Similarly, if one wishes to perform collaborative filtering on data from sites such as Amazon or NetFlix, the algorithms need to scale to tens of millions of users. Depending on the application, these domains could involve dimensionality equal to if not larger than the number of input vectors.

* Work done while at Google.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
WWW 2007, May 8-12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

One solution often applied to deal with data of this scale is to apply approximation techniques. While in theory many approximation techniques allow for making the probability of error negligibly small by tuning appropriate parameters, in practice these algorithms are applied in a manner that results in a non-trivial amount of error (e.g. see [4, 8].) Recent work from the database community [1, 21] on finding all similar pairs has focused instead on solving the problem exactly, and within the context of a database management system. We also propose an exact solution to the problem, though we ignore issues of DBMS integration and focus solely on performance issues. We show that a parsimonious indexing approach combined with several other subtle yet simple optimizations yield dramatic performance improvements. We validate our algorithms on a dataset comprised of publicly available data from the DBLP server, and on two real-world web applications: generating recommendations for the Orkut social network, and computing pairs of similar queries among the 5 million most frequently issued Google queries represented as vectors generated from their search result snippets.

Paper outline: We give a formal problem statement and define our terminology in Section 2. In Section 3 we describe related work. We present our algorithm in Section 4, where we limit attention to the cosine similarity metric. In some applications, the vector elements are binary; we therefore specialize the algorithms for this case. In the final part of this section, we extend the algorithms to disk resident data. We present the results of our empirical evaluation on large-scale memory and disk resident datasets in Section 5, and generalize our results to other similarity metrics in Section 6. In Section 7 we conclude with a summary of our contributions.

2. PROBLEM STATEMENT

Given a set of real-valued vectors $V = \{v_1, v_2, \ldots, v_n\}$ of fixed dimensionality m, a similarity function $\mathrm{sim}(x, y)$, and a similarity threshold t, we wish to compute the set of all pairs (x, y) and their similarity values $\mathrm{sim}(x, y)$ such that $x, y \in V$ and $\mathrm{sim}(x, y) \geq t$. We assume the similarity function is commutative. Thus, if the pair (x, y) meets the threshold, so does (y, x), and we need only include one in the result. We also assume vector values are non-negative.

For concreteness, we initially focus on the case of unit-length normalized vector input and cosine similarity. Cosine similarity is widely used, and produces high quality results across several domains [8, 9, 19, 20]. Restricted to unit-length input vectors x and y, cosine similarity is simply the vector dot product:

$$\mathrm{dot}(x, y) = \sum_i x[i] \cdot y[i]$$

For many problem domains, especially those involving textual data, input vectors are *sparse* in that a vast majority of vector weights are 0. A *sparse vector representation* for a vector x is the set of all pairs $(i, x[i])$ such that $x[i] > 0$ over all $i = 1 \ldots m$. We sometimes refer to such pairs as the *features* of the vector. The *size* of a vector x, which we denote as $|x|$, is the number of such pairs. Vector size should not be confused with vector length, or *magnitude*, which is typically denoted by $\|x\|$.

Given a set of sparse vectors V, an *inverted list* representation of the set consists of m lists $I_1, I_2, \ldots, I_m$ (one for each dimension), where list I_i consists of all pairs (x, w) such that vector x is in V, $x[i] = w$, and w is non-zero. Each I_i is a list and not a set since we may impose requirements on the order in which its elements appear.

For a given vector x we denote the maximum value $x[i]$ over all i as $\text{maxweight}(x)$. For a given dimension i, we denote the maximum value $x[i]$ over all vectors x in the dataset V as $\text{maxweight}_i(V)$.

3. RELATED WORK

The all-pairs similarity search problem is a generalization of the well-known nearest neighbor problem in which the goal is to find the nearest neighbors of a given point query. There is a wide body of work on this problem, with many recent works considering various approximation techniques [6, 10, 11, 12]. Approximation methods aim to reduce the dimensionality and/or size of the input vectors. The all-pairs similarity search problem has been directly addressed by Broder et al. in the context of identifying near-duplicate web pages [4]. In this work, the problem is solved approximately by applying a sketching function based on min-wise independent permutations in order to compress document vectors whose dimensions correspond to distinct n-grams (called *shingles*). To further improve performance, and to avoid issues of false resemblance, the implementation removes the most frequent shingles, though the actual effect of this heuristic is not quantified. Our methods do not resort to approximation or discarding of frequent features, and thus our contribution is to scale exact methods to larger datasets. Additionally, the techniques we propose are orthogonal to some approximation approaches that reduce vector size such as min hashing, and can be combined with them for even greater improvements in scalability.

Our work is also related to work in information retrieval (IR) optimization [5, 14, 15, 16, 17, 22, 23]. In IR, each document can be represented by a sparse vector of weighted terms, indexed through inverted lists. The user query, itself a list of terms, can be represented by a sparse vector of those terms with certain (usually equal) weights. Answering the query then amounts to finding all, or the top k, document vectors with non-zero similarity to the query vector, ranked in order of their similarity. The above-cited works propose various optimizations for this problem, which Turtle and Flood [23] classify into two categories: term-at-a-time and document-at-a-time approaches. Optimization strategies for document-at-a-time processing typically exploit the fact that the user is looking for the top r documents, and hence do not directly apply to our problem domain. The term-at-a-time strategies that are "safe" (complete), such as Turtle and Flood's term-at-a-time max_score optimization, are more applicable. Our work develops more powerful optimizations that exploit the particular requirements of the all-pairs similarity search problem.

The all-pairs similarity search problem has also been addressed in the database community, where it is known as the *similarity join* problem [1, 7, 21]. The techniques proposed in this work fall into two categories. Signature based solutions convert an imprecise matching problem to a precise matching problem followed by a filtering phase to eliminate false positives. Inverted list based solutions exploit IR techniques such as those discussed above. We leverage and extend upon some of the ideas proposed in this work, and use them as comparison points in our experiments.

Other related work includes clustering of web data [2, 8, 19, 20]. These applications of clustering typically employ relatively straightforward exact algorithms or approximation methods for computing similarity. Our work could be leveraged by these applications for improved performance or higher accuracy through less reliance on approximation.

4. ALGORITHMS

We now describe our algorithm that can be used to exactly (without approximation) solve the all-pairs similarity search problem. We present the algorithm as a series of refinements to simplify understanding of the various optimizations and engineering choices we made. We initially assume memory residence of the vectors and/or index structures, which allows each algorithm to solve the all-pairs problem with only a single pass over the data. Extensions for datasets that are larger than available memory appear in Section 4.6.

4.1 A Basic Inverted Index-Based Approach

One can naively solve the all-pairs similarity search problem by employing an IR system to build an inverted list index of the input vectors, and issuing each input vector as if it were a query to find the set of matching documents. Matching documents could then be filtered according to the similarity threshold, or the IR system itself could be modified to only return documents that meet the threshold. It is easy to see that such an algorithm produces correct results for any similarity function that must compare only those dimensions for which both inputs have non-zero weights. Unfortunately, there are several inefficiencies to this approach: (1) For each similar pair (x, y) it wastes computation effort also generating the pair (y, x). (2) It builds a full inverted list index prior to generating any output, and (3) it requires both the index and the vectors themselves remain memory resident for high performance.

A better approach is to instead build the index dynamically as suggested in [21], and to store the vector weights within the inverted index itself. Score accumulation [14] can then be applied to compute the similarity function using the inverted index structure alone. An important detail is exactly how the index is accessed in order to accumulate the similarity score. Previous works in IR [23] and similarity join processing [21] use a heap data structure to merge the inverted lists corresponding to each feature of a given vector x. Such an approach is appealing in IR when only the first n answers may be required, since only the initial portion of many lists may need to be accessed. However, in the case where all answers (meeting a particular score threshold) are required, each inverted list participating in the merge may often be scanned in its entirety. Our approach at exploiting the inverted lists is to instead scan each one individually and accumulate scores in a hash-based map. This approach offers improved locality and avoids the logarithmic overhead of the heap structure.

Pseudo-code for our map-based score accumulation approach, which we call All-Pairs-0, appears in Figure 1. The top-level function scans the dataset and incrementally builds the inverted lists. The Find-Matches-0 subroutine scans the inverted lists to perform score accumulation. We call any vector y that is added to the weight accumulation map a *candidate vector for* x, and the vector pair (x, y) a *candidate pair*. The algorithm checks the score of each candidate pair, and adds the pair in the result set if it meets the threshold.

4.2 Exploiting the Threshold During Indexing

Up to now we have consulted the similarity score threshold only to determine which candidate pairs make it into the result set. Previous work has described how to exploit a similarity threshold more aggressively in order to limit the set of candidate pairs that are considered [21, 23], but these approaches still involve building the complete inverted index over the vector input. Our approach is different than previous work in that rather than simply exploiting the threshold to reduce candidate pairs, we exploit the threshold to reduce the amount of information indexed in the first place. This approach dramatically reduces the number of candidate pairs

```
ALL-PAIRS-0(V, t)
  O ← ∅
  I_1, I_2, …, I_m ← ∅
  for each x ∈ V do
      O ← O ∪ FIND-MATCHES-0(x, I_1, …, I_m, t)
      for each i s.t. x[i] > 0 do
          I_i ← I_i ∪ {(x, x[i])}
  return O
```

```
FIND-MATCHES-0(x, I_1, …, I_m, t)
  A ← empty map from vector id to weight
  M ← ∅
  for each i s.t. x[i] > 0 do
      for each (y, y[i]) ∈ I_i do
          A[y] ← A[y] + x[i] · y[i]
  for each y with non-zero weight in A do
      if A[y] ≥ t then
          M ← M ∪ {(x, y, A[y])}
  return M
```

Figure 1. A basic inverted index based approach.

```
ALL-PAIRS-1(V, t)
→ Reorder the dimensions 1…m such that dimensions with
    the most non-zero entries in V appear first.
→ Denote the max. of x[i] over all x ∈ V as maxweight_i(V).
  O ← ∅
  I_1, I_2, …, I_m ← ∅
  for each x ∈ V do
      O ← O ∪ FIND-MATCHES-1(x, I_1, …, I_m, t)
→     b ← 0
→     for each i s.t. x[i] > 0 in increasing order of i do
→         b ← b + maxweight_i(V) · x[i]
→         if b ≥ t then
→             I_i ← I_i ∪ {(x, x[i])}
→             x[i] ← 0                ;; create x'
  return O
```

```
FIND-MATCHES-1(x, I_1, …, I_m, t)
  A ← empty map from vector id to weight
  M ← ∅
  for each i s.t. x[i] > 0 do
      for each (y, y[i]) ∈ I_i do
          A[y] ← A[y] + x[i] · y[i]
  for each y with non-zero weight in A do
→     ;; Recall that y' is the unindexed portion of y
→     s ← A[y] + dot(x, y')
      if s ≥ t then
          M ← M ∪ {(x, y, s)}
  return M
```

Figure 2. An algorithm that exploits the threshold during indexing.

considered and reduces overhead such as index construction and inverted list scanning during score accumulation.

The threshold-based refinement to our algorithm appears in Figure 2. The main loop of the All-Pairs function now iterates over features from most to least frequent, and avoids indexing any vector features until a particular condition is met. The result is to index just enough of the least frequent features to ensure that any vector y that has the potential of meeting the similarity threshold given x must be identified as a candidate of x during matching. The frequency-based feature ordering is not required for correctness; its effect is to heuristically minimize the length of the inverted lists.

By indexing only a subset of vector features, All-Pairs-1 reduces the overall number of candidate pairs that are considered by Find-Matches. Also note that vector features that are indexed are removed from the vector itself in order to preserve memory and speed up the computation that remains after accumulating weights from the partial index. Overall, each vector feature is stored only once, either within the partial index or within the vector itself.

Because of the fact that only a portion of each vector is indexed, after processing the inverted lists within Find-Matches, only a portion of the similarity score will be accumulated. Find-Matches-1 must therefore compute cosine similarity over the unindexed portion of any candidate vector in order to complete the similarity score computation. Even though explicit similarity computation over a pair of input vectors is required by this algorithm, it is almost always substantially faster than All-Pairs-0 because of the vast reduction in the maximum size of its inverted lists. We further optimize away this overhead in subsequent refinements.

To demonstrate correctness of this approach, we must establish two facts: (1) the algorithm correctly computes the similarity score between any candidate pair x and y, and (2) for the current vector x, any indexed vector y such that $\mathrm{dot}(x, y) \geq t$ must be produced as a candidate of x. Before we begin, recall that we denote the unindexed features from vector y as y'. Let us further denote the indexed features of vector y as y''. Now, to see why (1) holds, note that after accumulating scores over the indexed portion of y given x, we have accumulated in $A[y]$ the value $\mathrm{dot}(x, y'')$. Since $\mathrm{dot}(x, y) = \mathrm{dot}(x, y') + \mathrm{dot}(x, y'')$, the final sum in Find-Matches-1 (indicated by the arrow) computes the actual cosine similarity score.

To see why (2) holds, note that when indexing a vector, All-Pairs-1 maintains a trivial upperbound b on the score attainable by matching the first features of the current vector against any other vector in the dataset. As soon as this upperbound exceeds t, it begins indexing the remaining features. It thus follows that for any indexed vector y and vector x, we have that $\mathrm{dot}(x, y') < t$. We again note that $\mathrm{dot}(x, y) = \mathrm{dot}(x, y') + \mathrm{dot}(x, y'')$, hence for any vector x and indexed vector y for which $\cos(x, y) \geq t$, we have that $\cos(x, y'') > 0$. It follows that at least one indexed feature of y is in common with some feature of x, so y is produced as a candidate for x.

4.3 Exploiting a Specific Sort Order

Recall that the correctness of All-Pairs-1 relies on the fact that it indexes enough of each vector y to guarantee that y is produced as a candidate when matched against any other vector x in the dataset such that x and y satisfy the similarity threshold. Note however that correctness is maintained even under a more restrictive condition: a vector y must be produced as a candidate only when matched against those vectors that *follow it in the order in which the dataset is processed*. This suggests that by exploiting a particular dataset sort order, we may be able to further reduce the amount of features indexed, and consequently further improve algorithm performance. While previous work [21] has noted that sorting the dataset according to particular criteria such as vector size can improve algorithm performance due to the nature of the

```
ALL-PAIRS-2(V, t)
  Reorder the dimensions 1...m such that dimensions with
    the most non-zero entries in V appear first.
  Denote the max. of x[i] over all x ∈ V as maxweight_i(V).
→ Denote the max. of x[i] for i = 1...m as maxweight(x).
→ Sort V in decreasing order of maxweight(x).
  O ← ∅
  I_1, I_2, ..., I_m ← ∅
  for each x ∈ V do
      O ← O ∪ FIND-MATCHES-2(x, I_1, ..., I_m, t)
      b ← 0
      for each i s.t. x[i] > 0 in increasing order of i do
→         b ← b + min(maxweight_i(V), maxweight(x)) · x[i]
          if b ≥ t then
              I_i ← I_i ∪ {(x, x[i])}
              x[i] ← 0
  return O
```

Figure 3a. A version of All-Pairs that exploits a sort order of the input vectors to further reduce the amount of indexed data, and hence candidate pairs.

```
FIND-MATCHES-2(x, I_1, ..., I_m, t)
   1  A ← empty map from vector id to weight
   2  M ← ∅
→  3  remscore ← Σ_i x[i] · maxweight_i(V)
→  4  minsize ← t/maxweight(x)
   5  for each i s.t. x[i] > 0 in decreasing order of i do
→  6      Iteratively remove (y, w) from the front of I_i while |y| < minsize.
   7      for each (y, y[i]) ∈ I_i do
→  8          if A[y] ≠ 0 or remscore ≥ t then
   9              A[y] ← A[y] + x[i] · y[i]
→ 10      remscore ← remscore − x[i] · maxweight_i(V)
  11  for each y with non-zero weight in A do
→ 12      if A[y] + min(|y'|, |x|) · maxweight(x) · maxweight(y') ≥ t then
  13          s ← A[y] + dot(x, y')
  14          if s ≥ t then
  15              M ← M ∪ {(x, y, s)}
  16  return M
```

Figure 3b. A version of Find-Matches that directly exploits the similarity threshold.

data structures employed, this work did not otherwise exploit the sort order.

Figure 3a provides our refinement of the All-Pairs procedure that exploits a particular dataset sort order to further minimize the number of indexed features. The ordering over the vectors guarantees any vector x that is produced after a vector y has a lower maximum weight. Significantly fewer features of each vector can now be indexed while still guaranteeing candidacy of a vector when matched against those that follow it in the iteration order. To show why, we apply an argument analogous to the correctness claim we made for All-Pairs-1. Note that All-Pairs-2 now maintains an upperbound b on the score attainable by matching the first features of a vector against any other vector that follows it in the dataset. As before, as soon as this upperbound exceeds t it begins indexing the remaining features. Thus, for any indexed vector y and vector x that follows it in the dataset ordering, we have that $\mathrm{dot}(x, y') < t$, from which correctness follows.

4.4 Exploiting the Threshold During Matching

The modifications we made to Find-Matches in Find-Matches-1 indirectly exploit the similarity threshold to reduce candidate pairs by virtue of the partially indexed vectors, but the threshold can be exploited more directly in order to further improve performance. Our final refinement of Find-Matches (Figure 3b) exploits the threshold directly in three ways. The first two ways we describe do not rely on the dataset sort order, while the third does.

The first method of exploiting the threshold relies on the fact that as we iterate over the features of x, we eventually get to the point where if a vector has not already been identified as a candidate of x, then there is no way it can meet the score threshold. Find-Matches-2 therefore keeps track of such a point by maintaining an upperbound *remscore* on the score attainable between x and any vector y that shares none of the "already processed" features (lines 3 & 10). Once the upperbound drops beneath the score threshold, the algorithm switches to a phase where it avoids putting new candidates into the map, and instead only updates accumulated scores of those vectors already in the map (line 8).

The second method of exploiting the threshold takes place in the candidate iteration loop. Note now that Find-Matches-2 does not unconditionally compute the dot product over x and each partial candidate vector y'. Instead, it computes a cheap upperbound on $\mathrm{dot}(x, y)$, and only explicitly computes the dot product should the upperbound meet the threshold (line 12). With this optimization,

```
ALL-PAIRS-BINARY(V, t)
  Reorder the dimensions 1...m such that dimensions with
    the most non-zero entries in V appear first.
  Sort V in increasing order of |x| .
  O ← ∅
  I_1, I_2, ..., I_m ← ∅
  for each x ∈ V do
      O ← O ∪ FIND-MATCHES-BINARY(x, I_1, ..., I_m, t)
      b ← 0
      for each i s.t. x[i] = 1 in increasing order of i do
          b ← b + 1
→         if b/|x| ≥ t then
              I_i ← I_i ∪ {x}
              x[i] ← 0
  return O
```

```
FIND-MATCHES-BINARY(x, I_1, ..., I_m, t)
  A ← empty map from vector id to int
  M ← ∅, remscore ← |x|
→ minsize ← |x| · t^2
  for each i s.t. x[i] = 1 in decreasing order of i do
→     Remove all y from I_i s.t. |y| < minsize .
      for each y ∈ I_i do
          if A[y] ≠ 0 or remscore ≥ minsize then
              A[y] ← A[y] + 1
      remscore ← remscore − 1
  for each y with non-zero count in A do
      if (A[y] + |y'|) / (√|x| · √|y|) ≥ t then
          d ← (A[y] + dot(x, y')) / (√|x| · |y|)
          if d ≥ t then
              M ← M ∪ {(x, y, d)}
  return M
```

Figure 4. The All-Pairs algorithm specialized for binary vector input.

most candidates of x are quickly excluded without iterating over their (potentially numerous) unindexed features.

The third optimization exploits the fact that for a vector x, we can show that any vector y that satisfies the similarity score threshold must meet the following minimum size ("minsize") constraint:

$$|y| \geq t / \text{maxweight}(x)$$

To see why, by definition, if two vectors x and y meet the similarity score threshold then $\text{dot}(x, y) \geq t$. Now, note that $\text{dot}(x, y) < \text{maxweight}(x) \cdot |y|$. Thus, we have that $\text{maxweight}(x) \cdot |y| \geq t$, from which the claim follows.

Now, because vectors are processed by All-Pairs-2 in decreasing order of $\text{maxweight}(x)$, the minimum allowable matching vector size increases monotonically as the algorithm progresses. We can therefore safely and permanently remove from the index any vector encountered that fails to meet the size constraint at any given point. We remove such vectors lazily during Find-Matches at the time a particular inverted list is required for score accumulation (line 6). Note that for efficiency we only remove such vectors from the beginning of each inverted list, which implies we do not necessarily remove all vectors that fail meet the size constraint. But note that the inverted lists are themselves grown in decreasing order of maxweight, and that maxweight is highly inversely correlated with vector size. The beginning of each list will thus typically contain most of the vectors eligible for removal.

In implementations (such as ours) that use arrays for inverted lists, removing elements from the beginning of a list can be costly due to the shifting of the remaining elements. In such a case, one can instead maintain a start offset into each inverted list array that can be incrementally advanced over vectors that no longer meet the size constraint. If desired, inverted lists could still be periodically filtered in order to reclaim memory occupied by the vectors too small to meet the similarity threshold.

4.5 Specializations for Binary Vector Data

Many applications have data that is most naturally represented using binary valued sparse vectors. While one could directly apply All-Pairs to binary vector data by simply normalizing the inputs to unit length and operating on the resulting weighted vectors, specialization avoids the need to perform any such conversion, and also allows for other optimizations. For example, when the input vectors are binary, there is no need to store vector weights within the inverted index as long as we can get at the vector sizes in order to appropriately compute the similarity function. In the case of cosine similarity, the similarity between two non-normalized binary valued vectors is as follows:

$$\cos(x, y) = \frac{\sum_i x_i \cdot y_i}{\sqrt{|x|} \cdot \sqrt{|y|}} = \frac{\text{dot}(x, y)}{\sqrt{|x|} \cdot \sqrt{|y|}}$$

Because binary input vectors are no longer of unit length, instead of sorting the input and computing bounds based on the maxweight vector property, the property of interest becomes the size of the input vectors. Recall that a vector's size is simply its number of features, and hence after unit-length normalization of binary valued vectors, the maximum weight of a vector is inversely proportional to its size. The binary-valued input specialization of All-Pairs algorithm appears in Figure 4.

Correctness of the reduced indexing condition used in All-Pairs-Binary follows from the fact that the algorithm indexes all but a number of features b of a given vector y such that $b / |y| < t$. Recall that we denote the unindexed and indexed features of y and y' and y'' respectively. We show that if vectors x and y satisfy the similarity threshold, they share at least one indexed term. Note first that since $|x| \geq |y|$:

$$\frac{\text{dot}(x, y')}{\sqrt{|x|} \cdot \sqrt{|y|}} \leq \frac{b}{\sqrt{|x|} \cdot \sqrt{|y|}} \leq \frac{b}{\sqrt{|y|} \cdot \sqrt{|y|}} = \frac{b}{|y|} < t$$

Next, since $\cos(x, y) = \frac{\text{dot}(x, y')}{\sqrt{|x|} \cdot \sqrt{|y|}} + \frac{\text{dot}(x, y'')}{\sqrt{|x|} \cdot \sqrt{|y|}}$ and $\cos(x, y) \geq t$, we must have that:

$$\frac{\text{dot}(x, y'')}{\sqrt{|x|} \cdot \sqrt{|y|}} > 0$$

Hence x and y share at least one indexed term.

We have highlighted the minsize index pruning optimization used by Find-Matches-Binary, since it has been strengthened for the case of binary data. First, note that since inverted lists are grown in increasing order of vector size, all vectors that fail to meet the minimum size constraint appear at the front of the list. All such vectors can therefore be efficiently removed or advanced over, instead of just most of them as was the case with weighted

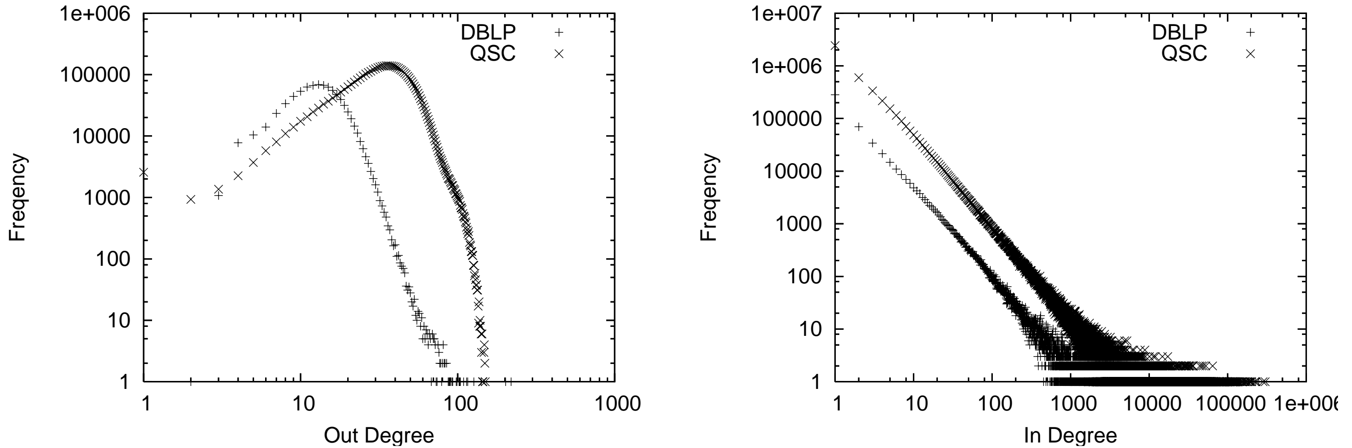


Figure 5. Outdegree and Indegree distributions of the DBLP and query snippet containment data.

vector data. Now, for any pair of vectors x and y that meet a cosine similarity threshold t, we exploit the following minsize constraint, which as before increases monotonically as the algorithm progresses:

$$|y| \geq |x| \cdot t^2$$

To see why this condition holds, we have by definition:

$$\frac{\mathrm{dot}(x, y)}{\sqrt{|x|} \cdot \sqrt{|y|}} \geq t$$

and hence,

$$\frac{\mathrm{dot}(x, y)^2}{|y|} \geq |x| \cdot t^2$$

Since $\mathrm{dot}(x, y) \leq |y|$, we must also have that

$$\frac{|y|^2}{|y|} = |y| \geq |x| \cdot t^2$$, and the condition is established.

4.6 Extensions for Disk Resident Data

We have described All-Pairs as storing each feature in RAM at most once during its execution. For datasets whose features exceed available memory, virtual memory thrashing would make such an approach impractical. For such situations, we have developed an out-of-core variant of the All-Pairs algorithm. For starters, All-Pairs must use an out-of-core sorting algorithm to impose the desired vector ordering. The remaining problem is to prevent the various memory resident data structures used for vector matching from exceeding available memory. Our solution is to have the algorithm index newly encountered vectors only up to the point where further indexing would exceed available memory. At this point, the algorithm switches to a "matching only" phase in which each remaining vector is kept in memory only for the duration of the Find-Matches call. To ensure completeness, once the end of the dataset is reached during the matching only phase, the algorithm clears out the index and other memory resident vector features, and performs a seek into the dataset to resume indexing anew from the last point it previously stopped indexing. This strategy is conceptually similar to the well-known block nested loops strategy employed by database systems for out-of-core table joins [18]. Due to the nature in which vectors are removed from the index during matching, it is possible for the partial index to become completely empty. Should this happen, our implementation immediately terminates the current dataset scan and proceeds to indexing and matching against the next dataset block.

5. EXPERIMENTS

In this section we compare All-Pairs to previous inverted list [21] and signature based methods [1, 11]. We run these algorithms on three real world datasets which we describe in Section 5.1. Two of these datasets come from web applications, while the third is a much smaller but publicly available dataset compiled from DBLP data. All our implementations are in C++, and use the standard template library vector class for inverted lists and most other structures. We used the Google dense_hash_map class for performing score accumulation in Find-Matches and the dense_hash_set class for signature matching (both freely available from http://code.google.com/). All experiments in this subsection were performed on a 3.4 GHz Pentium-4 class machine with 3 Gbytes of RAM and a 7200 RPM SATA-IDE hard drive.

5.1 Datasets

QSC: The first dataset we used was motivated by applications of query snippet data to determine semantic similarity between queries [19] and for taxonomy generation [9]. We selected roughly the 5 million most frequent queries that were submitted to the Google search engine from the United States during one week in January 2006. Queries in the list were then normalized (e.g. excessive whitespace stripped). We then used the Google SOAP search API (http://www.google.com/apis/) to generate the top-20 search results and their snippet data for each query. We used the set of queries as candidate terms in the vector for each query, i.e., if query x_j appeared in the snippets for query x_i (and also met the criteria described next), then $x_i[j] = 1$. We only included term x_j in the vector for x_i if x_j appeared at least twice in x_i's snippets, and the frequency of x_j in x_i's snippets was significantly higher than the background frequency of x_j across all queries.

The indegree (feature frequency) and outdegree (features per vector) distributions for this particular *query snippet containment* (QSC) dataset appear in Figure 5. While our significance thresholding resulted in a relatively restrained outdegree with an average of 36, the indegree follows a power law distribution. This dataset fits entirely in a 2 gigabyte process space, so the runtimes we report for it are almost entirely dominated by CPU overhead.

Orkut: The second dataset is the Orkut (http://orkut.com/) social network, in which each user is represented by a binary vector over the space of all users. A user's vector has a 1 in any dimension that represents himself or anyone the user has listed as a "friend." The Orkut graph is undirected since friendship is treated as a symmetric relationship. It consists of almost 20 million nodes (vectors) and 2 billion links (non-zero weights), yielding roughly

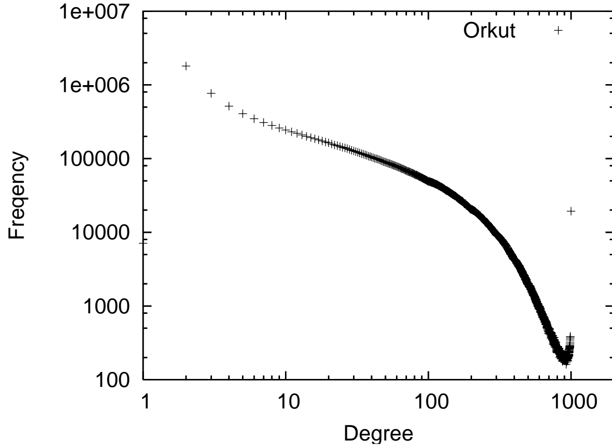


Figure 6. Degree distribution of the Orkut social network.

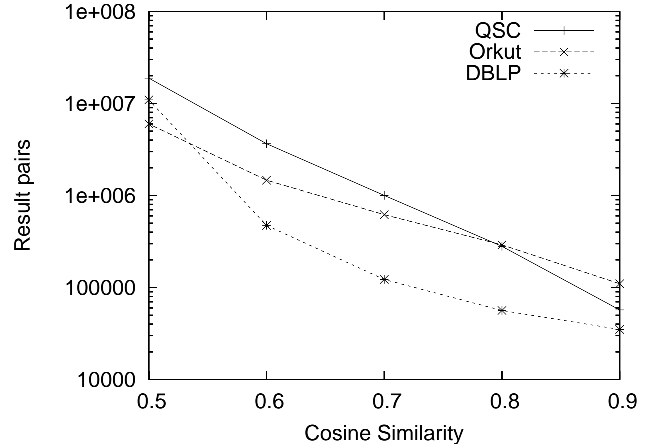


Figure 7. Algorithm output size for a given threshold.

100 features per vector on average. Similar pairs from this data reflect users who have a significant overlap in friends. This similarity data could be used in a variety of ways. For example, users with high similarity scores who are not already friends may be good targets for introductions.

The degree distribution for the Orkut graph appear in Figure 6. As can be clearly seen from this figure, Orkut limits the number of friends of each user to 1000. This dataset is too large to fit into main memory on our test machine, so all experimental results on this dataset report the wall-clock time of the out-of-core algorithm variants. These variants were configured to halt indexing and enter the matching only phase after loading 250 million non-zero weights. Each algorithm thus required 8 passes over the dataset to process it in its entirety.

DBLP: The third dataset is compiled from a snapshot of the DBLP data as described in [1]. Our snapshot consisted of almost 800,000 vectors and 14 features per vector on average. The total number of dimensions was roughly 530,000.

All datasets are sorted in increasing order of vector size. Runtimes are for the sorted dataset even if the algorithm does not explicitly exploit the sort order. We confirmed the finding from previous work [21] that performance was always better on the sorted dataset. We did not include the time required to perform the dataset sort, which for QSC and DBLP was negligible. Because Orkut required an out-of-core sort (the linux "sort" command was used with a 2 gigabyte memory buffer), sorting required 21 minutes.

We use similarity thresholds between .5 and .9 in our experiments. We chose .5 as the lowest setting based on previous work on semantic query similarity that found semantically similar query pairs to typically have scores above .5 [19]. The number of result pairs for each threshold and each dataset appears in Figure 7.

5.2 Comparison of Inverted List-Based Approaches

We compared All-Pairs to another inverted-list based approach, ProbeOpt-sort [21]. ProbeOpt-sort is an algorithm that dynamically builds an inverted index and consults the index to identify candidate vectors much like All-Pairs. Unlike All-Pairs, it indexes every feature, but it does use the similarity threshold to reduce the inverted lists that must be scanned in full in order to generate candidates. For each candidate identified by the inverted list scans, any remaining relevant inverted lists are probed using a doubling binary search. Before each binary search, a score upperbound is computed, and if the score upperbound falls beneath the threshold, the algorithm moves on to the next candidate. ProbeOpt-sort was presented in [21] for the Overlap similarity metric, so for an apples-to-apples comparison, we modified our implementation for cosine distance by having ProbeOpt-sort apply the same cosine distance upperbounding technique used by All-Pairs-Binary within its candidate evaluation loop. We did not compare against a variant of ProbeOpt-sort called ProbeOpt-cluster since ProbeOpt-cluster was found to be only slightly faster.

To shed more insight on the impact of the optimizations within All-Pairs, we implemented a version of the algorithm which, like ProbeOpt-sort, did not explicitly rely on the dataset sort order. The algorithm is called All-Pairs-Unsorted within the graphs. The difference between All-Pairs-Unsorted and All-Pairs is that All-Pairs-Unsorted does not explicitly exploit the sort order of the records (even though we still run it on the sorted datasets.) So All-Pairs-Unsorted uses only $\text{maxweight}_i(V)$ and not $\text{maxweight}(x)$ to determine how much of the vector needs to be indexed. For binary datasets, the reduced indexing condition of All-Pairs-Unsorted is $b/|v| < t^2$ instead of $b/|v| < t$. All-Pairs-Unsorted also cannot exploit the minsize pruning method that iteratively removes small records from the front of the inverted lists.

Results for all three datasets appear in Figure 8. Even though it lacks any explicit sort-order-dependent optimizations, All-Pairs-Unsorted always outperforms ProbeOpt-sort, typically by at least a factor of 2. Much of the speedup is due to indexing only part of each input vector, though hash-based score accumulation in place of queue-based merging is also a factor. Note that for all datasets, the sort-dependent optimizations lead to an even larger performance gain. On the QSC dataset, All-Pairs outperforms ProbeOpt-sort from a factor of 22x at .9 threshold to a factor of 8x at .7. ProbeOpt-sort failed to complete at lower threshold settings within the 1000 minute time limit we imposed.

On the Orkut dataset, All-Pairs is much faster (2-3x) than All-Pairs-Unsorted, but the performance difference is less than witnessed on the other datasets for the following reasons: (1) The artificial 1000 friend limit prevents highly frequent features -- a key source of inefficiency, and (2) the fact that the data was disk-resident led to some IO overhead, which was similar across all algorithms. Our ProbeOpt-sort variant modified to handle out-of-core datasets based on the approach described Section 4.6 was not able to complete within 1000 minutes even with a .9 similarity threshold. We suspect ProbeOpt-sort performed poorly on this dataset due to the longer average vector size.

Performance characteristics on the DBLP data was similar to QSC, with All-Pairs providing a speedup of approximately 6x over ProbeOpt-Sort across all similarity thresholds.

5.3 Comparison with Signature-Based Methods

Here we compare All-Pairs with two signature-based methods for finding all similar pairs: LSH [11] and PartEnum [1]. LSH

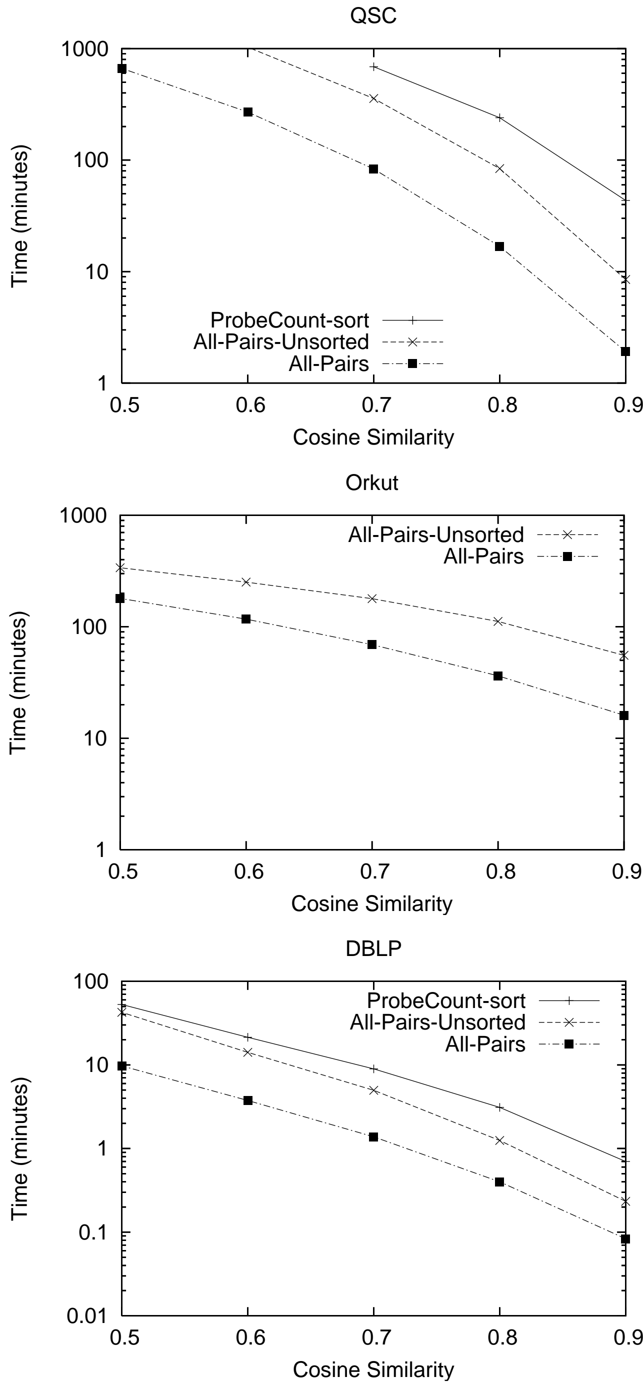


Figure 8. Inverted List Algorithm Performance

(Locality Sensitive Hashing) is a well-known approximate algorithm for the problem. It first generates d random projections of the set of dimensions. Next, for each input vector and each random projection, the algorithm generates a signature by combining the vector features belonging to that projection. Thus d signatures are generated for each vector. If two vectors have a matching signature along the same projection, a candidate pair is generated. The false negative rate is controlled using the number of projections d. In our experiment, we set d to allow for at most 5% false negatives measured as actual error.

PE (PartEnum) is another signature based algorithm that like All-Pairs is guaranteed to generate all pairs that satisfy the similarity threshold. The intuition is that a similarity threshold t can be converted to a threshold k on the Hamming distance between two vectors. Signatures are generated in such that any two vectors whose Hamming distance is less than k will have at least one matching signature. The conversion of Jaccard similarity to Hamming distance is covered in [1]:

$$k = \frac{(1-t)}{(1+t)} \cdot (|x| + |y|)$$

To use PartEnum with the cosine similarity metric, we derived the following conversion from cosine similarity to Hamming distance:[1]

$$k = |x| + |y| - 2 \cdot t \cdot \sqrt{|x| \cdot |y|}$$

Given a vector x, an upper bound l_u on the size of any vector that matches x with cosine similarity at least t is as follows:

$$l_u = |x| / t^2$$

We use this bound to partition the vectors into groups by size, such that only the vectors in each group or in adjacent groups need to be compared. Since we follow the same approach as in Figure 6 of [1], we omit further details, except to point out that the equations corresponding to lines (b) and (c) in Figure 6 for cosine similarity are: $r_i = \lfloor l_i / t^2 \rfloor$ in line (b), and $k_i = \lfloor 2 \cdot (1-t) \cdot r_i \rfloor$ in line (c).

Figure 9 shows the running times for PartEnum, LSH and All-Pairs on the DBLP data using both cosine similarity (left graph) and Jaccard similarity (right graph). With cosine similarity, PartEnum fails to complete for all but the highest 2 threshold settings within the 200 minute time limit we imposed; at these settings, All-Pairs is 16 to 700 times faster. The speedup given Jaccard similarity is less dramatic, but still at least an order of magnitude. The reason for the wider performance discrepancy when using cosine distance is that the bound on Hamming distance is a factor of $1 + t$ higher. Without tight bounds on Hamming distance, PartEnum needs to generate many candidates to guarantee completeness. Even though LSH only provides an approximate solution to the problem, All-Pairs outperforms it by 2 to 15 times for cosine similarity, and 1.3 to 6 times for Jaccard similarity. This discrepancy in speedups is due to the fact that at the same threshold, LSH needs to use more signatures for the cosine similarity than Jaccard similarity to ensure a 5% or lower false negative rate.

To understand the performance advantage of All-Pairs over the signature based schemes, note that signature based algorithms fully scan each vector in a candidate pair to compute their similarity score. So for example if a vector x appears in 10 candidate pairs, x will be scanned in its entirety 10 times. In contrast, All-Pairs only scans x once for all vectors shorter than x. For vectors y longer than x, only the portions of x' (the indexed part of x) that intersect y are scanned. The unindexed portion of x is rarely scanned due to the effectiveness of the score bound condition. All-Pairs also exhibits better cache locality, since it sequentially scans inverted lists, rather than jumping between candidate pairs. Finally, and perhaps most importantly, signature based schemes are less effective in reducing the number of candidate pairs at lower

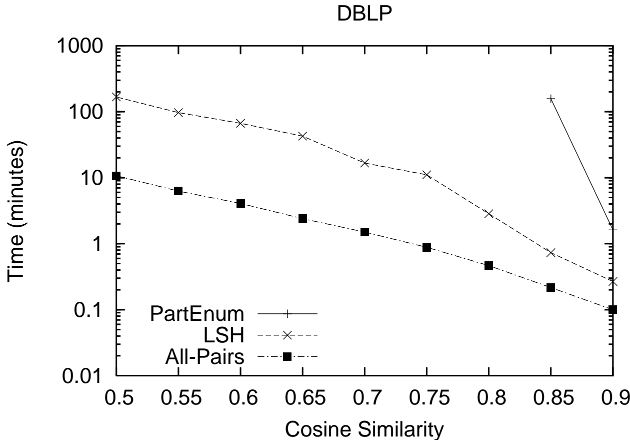
---

[1] We give the derivation for completeness. Hamming distance is defined as $|x| + |y| - 2 \cdot \mathrm{dot}(x, y)$. Given vectors x and y that satisfy a cosine similarity threshold t:

$$\cos(x, y) = \mathrm{dot}(x, y) / (\sqrt{|x| \cdot |y|}) \geq t \text{ , and thus:}$$

$$\mathrm{dot}(x, y) \geq t \cdot \sqrt{|x| \cdot |y|} \text{ . Finally, we get:}$$

$$|x| + |y| - 2 \cdot \mathrm{dot}(x, y) \leq |x| + |y| - 2 \cdot t \cdot \sqrt{|x| \cdot |y|}$$



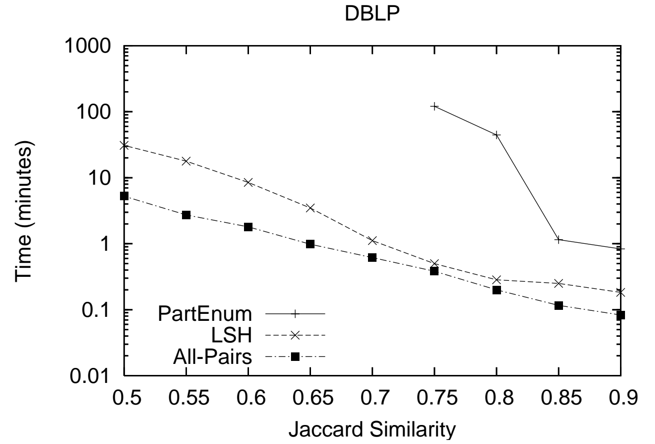


Figure 9. Signature-Based Algorithm Performance

similarity thresholds. It is also worth noting that both PartEnum and LSH performance are highly dependent on their parameter settings. We spent considerable time tuning parameters to optimize their runtime.

6. OTHER SIMILARITY MEASURES

Although our main algorithm explicitly exploits the cosine similarity metric, it can be generalized to several other similarity measures. We demonstrate this generalization capability with the following common similarity measures: Jaccard coefficient, dice coefficient, and overlap coefficient. These similarity measures are defined over binary vector inputs as follows:

Cosine	$\mathrm{sim}(x, y) = \frac{\mathrm{dot}(x, y)}{\sqrt{\|x\| \cdot \|y\|}}$
Overlap	$\mathrm{sim}(x, y) = \frac{\mathrm{dot}(x, y)}{\min(\|x\|, \|y\|)}$
Dice	$\mathrm{sim}(x, y) = \frac{2 \cdot \mathrm{dot}(x, y)}{\|x\| + \|y\|}$
Jaccard	$\mathrm{sim}(x, y) = \frac{\mathrm{dot}(x, y)}{\|x\| + \|y\| - \mathrm{dot}(x, y)}$

Table 1: Similarity Measures

Conveniently, the reduced indexing condition used in All-Pairs-Binary already preserves completeness given any of these similarity measures. Recall that the algorithm indexes all but a number of features b of a given vector y such that $b/|y| < t$. We therefore need only show that the unindexed portion of the vector y (denoted y'), which consists of the last b unit valued weights in y, contributes an amount to the similarity score that alone is not enough to meet the threshold. For an indexed vector y and any vector x following it in the ordering ($|x| \geq |y|$), the following inequalities establish these results.

Overlap coefficient:

$$\frac{\mathrm{dot}(x, y')}{\min(|x|, |y|)} \leq \frac{b}{\min(|x|, |y|)} = \frac{b}{|y|} < t$$

Dice coefficient:

$$\frac{2 \cdot \mathrm{dot}(x, y')}{|x| + |y|} \leq \frac{2b}{|y| + |y|} = \frac{b}{|y|} < t$$

Jaccard coefficient:

$$\frac{\mathrm{dot}(x, y')}{|x| + |y| - \mathrm{dot}(x, y)} \leq \frac{b}{|x| + |y| - \mathrm{dot}(x, y)}$$

Note that $\mathrm{dot}(x, y) \leq |x|$, hence:

$$\frac{b}{|x| + |y| - \mathrm{dot}(x, y')} \leq \frac{b}{|x| + |y| - |x|} = \frac{b}{|y|} < t$$

Given that the existing reduced indexing condition is sufficient, the only modifications required are of Find-Matches-Binary where we compute the similarity score, upperbound the similarity score, and compute a monotone minimum size constraint on candidate vectors. We forego providing details to these variations due to space constraints, though they can be derived by appropriately adapting the techniques used to handle cosine distance. Similar generalizations are also possible for other similarity functions over weighted vector data.

Performance of the All-Pairs algorithm when using these alternate similarity metrics appears in the left-hand graph of Figure 10. Note that there is a modest increase in runtime for some of these alternate metrics. The explanation for this runtime increase can be seen from the accompanying graph which reports the number of similar pairs in the output. For the same threshold setting, the overlap measure allows many more pairs, which results in an increase in the number of vector intersections that must be performed by Find-Matches. Dice is slower than cosine distance despite producing similar size output because the minsize of matching vectors cannot be bounded as effectively.

7. CONCLUSIONS AND FUTURE WORK

The problem of finding all pairs of similar vectors arises in many applications such as query refinement for search engines and collaborative filtering. We presented two key insights in this paper not identified in previous work:

- An inverted list based approach to the problem need not build a complete inverted index over the vector input.
- Appropriately ordering the vectors in addition to the dimensions can vastly reduce the search space.

We aggressively exploited these insights to produce an algorithm, All-Pairs, that is easy to implement and does not require any parameter tuning. The simplicity of the approach is combined with dramatic performance gains: We found All-Pairs to be 5 to 22 times faster than ProbeCount-sort, and between 10 to over 700 times faster than PartEnum. All-Pairs is even 1.3 to 15 times faster than an approximate algorithm, LSH tuned to have a 5% false negative rate.

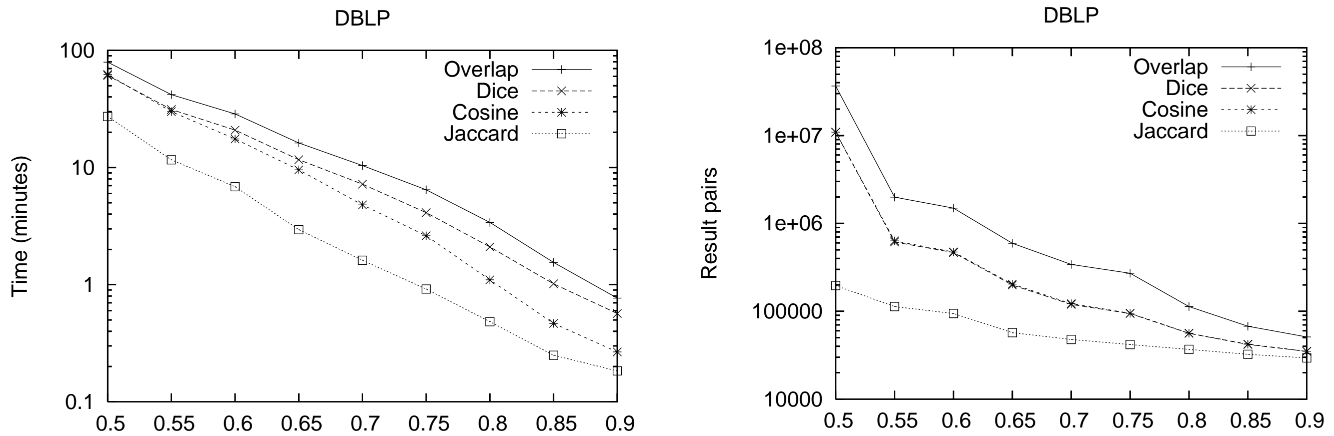


Figure 10. All-Pairs performance with alternate similarity functions.

While we have focused primarily on scalability with respect to the number of records in sparse data, we believe scalability of all-pairs similarity search algorithms with respect to feature density merits further study. Finally, in addition to developing further performance improvements for the all-pairs similarity search problem, we believe future work might address how algorithms for the problem can serve as useful primitives for other mining tasks. Interesting possibilities include exploiting all similar pairs for improving the quality of heuristic clustering approaches, performing deeper social network analysis, or in improving performance of related problems [3].

8. ACKNOWLEDGEMENTS

We thank Ellen Spertus for her assistance with the Orkut data, and Tim Heilman for his assistance with generating datasets for semantic query similarity.

9. REFERENCES

[1] A. Arasu, V. Ganti, & R. Kaushik (2006). Efficient Exact Set-Similarity Joins. In *Proc. of the 32nd Int'l Conf. on Very Large Data Bases*, 918-929.

[2] D. Beeferman & A. Berger (2000). Agglomerative Clustering of a Search Engine Query Log. In *Proc. of the 6th ACM-SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, 407-416.

[3] C. Böhm, B. Braunmüller, M. Breunig, & H.-P. Kriegel (2000). High Performance Clustering Based on the Similarity Join. In *Proc. of the 2000 ACM CIKM International Conference on Information and Knowledge Managemen*t, 298-305.

[4] A. Z. Broder, S. C. Glassman, M. S. Manasse, & G. Zweig (1997). Syntactic clustering of the Web. In *Proc. of the 6th Int'l World Wide Web Conference*, 391-303.

[5] C. Buckley & A. F. Lewit (1985). Optimization of Inverted Vector Searches. In *Proc. of the Eight Annual Int'l Conf. on Research and Development in Information Retrieval*, 97-110.

[6] M. S. Charikar (2002). Similarity Estimation Techniques from Rounding Algorithms. In *Proc. of the 34th Annual Symposium on Theory of Computing*, 380-388.

[7] S. Chaudhuri, V. Ganti, & R. Kaushik (2006). A Primitive Operator for Similarity Joins in Data Cleaning. In *Proc. of the 22nd Int'l Conf on Data Engineering,* 5.

[8] S. Chien & N. Immorlica (2005). Semantic Similarity Between Search Engine Queries Using Temporal Correlation. In *Proc. of the 14th Int'l World Wide Web Conference*, 2-11.

[9] S.-L. Chuang & L.-F. Chien (2005). Taxonomy Generation for Text Segments: A Practical Web-Based Approach. In *ACM Transactions on Information Systems*, 23(4), 363-396.

[10] R. Fagin, R. Kumar, & D. Sivakumar (2003). Efficient Similarity Search and Classification via Rank Aggregation. In *Proc. of the 2003 ACM-SIGMOD Int'l Conf. on Management of Data*, 301-312.

[11] A. Gionis, P. Indyk, & R. Motwani (1999). Similarity Search in High Dimensions via Hashing. In *Proc. of the 25th Int'l Conf. on Very Large Data Bases*, 518-529.

[12] P. Indyk, & R. Motwani (1998). Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proc. of the 30th Symposium on the Theory of Computing*, 604-613.

[13] A. Metwally, D. Agrawal, & A. El Abbadi (2007). DETECTIVES: DETEcting Coalition hiT Inflation attacks in adVertising nEtworks Streams. In *Proc. of the 16th Int'l Conf. on the World Wide Web*, to appear.

[14] A. Moffat, R. Sacks-Davis, R. Wilkinson, & J. Zobel (1994). Retrieval of partial documents. In *The Second Text REtrieval Conference,* 181-190.

[15] A. Moffat & J. Zobel (1996). Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349-379.

[16] M. Persin (1994). Document filtering for fast ranking. In *Proc. of the 17th Annual Int'l Conf. on Research and Development in Information Retrieval,* 339-348.

[17] M. Persin, J. Zobel, & R. Sacks-Davis (1994). Fast document ranking for large scale information retrieval. In *Proc. of the First Int'l Conf. on Applications of Databases, Lecture Notes in Computer Science v819*, 253-266.

[18] R. Ramakrishnan & J. Gehrke (2002). *Database Management Systems*. McGraw-Hill; 3rd edition.

[19] M. Sahami & T. Heilman (2006). A Web-based Kernel Function for Measuring the Similarity of Short Text Snippets. In *Proc. of the 15th Int'l Conf. on the World Wide Web*, 377-386.

[20] E. Spertus, M. Sahami, & O. Buyukkokten (2005). Evaluating Similarity Measures: A Large Scale Study in the Orkut Social Network. In *Proc. of the 11th ACM-SIGKDD Int'l Conf. on Knowledge Discovery in Data Mining*, 678-684.

[21] S. Sarawagi & A. Kirpal (2004). Efficient Set Joins on Similarity Predicates. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, 743-754.

[22] T. Strohman, H. Turtle, & W. B. Croft (2005). Optimization Strategies for Complex Queries. In *Proc. of the 28th Annual Int'l ACM-SIGIR Conf. on Research and Development in Information Retrieval,* 219-225.

[23] H. Turtle & J. Flood (1995). Query Evaluation: Strategies and Optimizations. In *Information Processing & Management*, 31(6), 831-850.