Incremental All Pairs Similarity Search for Varying Similarity Thresholds

Amit Awekar, Nagiza F. Samatova, and Paul Breimyer
acawekar@ncsu.edu, samatovan@ornal.gov, pwbreimy@ncsu.edu
North Carolina State University, Raleigh, NC
Oak Ridge National Laboratory, Oak Ridge, TN

***Abstract*— All Pairs Similarity Search ($APSS$) is a ubiquitous problem in many data mining applications and involves finding all pairs of records with similarity scores above a specified threshold. In this paper, we introduce the problem of *Incremental All Pairs Similarity Search* ($IAPSS$), where $APSS$ is performed multiple times over the same dataset by varying the similarity threshold. To the best of our knowledge, this is the first work that addresses the $IAPSS$ problem. All existing solutions for $APSS$ perform redundant computations by invoking $APSS$ independently for each threshold value.**

In contrast, our solution to the $IAPSS$ problem avoids redundant computations by storing the history of previous $APSS$ invocations and using index splitting. While offering obvious benefits, the computation and I/O intensive nature of the $IAPSS$ solution raises two key research challenges: (1) to develop efficient I/O techniques to manage computation history and (2) to efficiently identify and prune redundant computations. We address these challenges through the proposed (a) history binning technique that clusters record pairs based on similarity values and performs I/O during the similarity computation, and (b) splitting of inverted index that maps each dimension to a list of records that have a non-zero projection along that dimension. As a result, we evaluate the effectiveness of our techniques by demonstrating speed-ups in the order of $2X$ to over 10^5X over the state-of-the-art $APSS$ algorithm for four real-world large-scale datasets.

I. Introduction

Many data mining techniques search for all pairs of records that have similarity scores above a specified threshold [4], [13]. In the literature, this problem is referred to as *similarity join* [13] or *all pairs similarity search* (APSS) [4]. For example, the Jarvis-Patrick algorithm for clustering sparsifies the similarity score matrix by retaining only those entries that satisfy a predefined threshold [6].

Selecting a meaningful similarity threshold is an art because it is data dependent. Domain experts often use a trial and error approach by looking at the quality of output. For example, the optimal threshold for sparsifying the similarity score matrix in the Jarvis-Patrick algorithm can be determined only after evaluating the quality of different clusterings by varying the similarity threshold for sparsification.

Varying the similarity threshold leads to another important problem that we refer to as the *incremental all pairs similarity search* ($IAPSS$), which performs $APSS$ multiple times on the same dataset by varying the similarity threshold value. The $IAPSS$ problem is challenging to solve when it is applied frequently or over large datasets.

However, there are a number of important applications that require efficient handling the $IAPSS$ problem. For example, the output of $IAPSS$ is used to detect all near duplicate document pairs [13]. A news search engine has to solve the $IAPSS$ problem every few minutes over a small subset of the web, whereas a general web search engine has to solve the $IAPSS$ problem once every few days, but over the entire web.

To the best of our knowledge, the $IAPSS$ problem has not received a special treatment in the literature and the “brute-force” strategy is used instead. Namely, applying a new instance of $APSS$ after each similarity threshold value changes. Obviously, this solution may be inefficient due to inherent redundancies.

All of the existing solutions for $APSS$ [13], [4], [2], [9] do not exploit the fact that a significant part of the computation is redundant across multiple invocations of $APSS$, because each of the $APSS$ instances executes independently for changing similarity threshold values. For example, consider performing $APSS$ twice on a dataset. Initially, the threshold value is 0.9 and later it is reduced to 0.8. All pairs present in the output of the first $APSS$ will also exist in the output of the second $APSS$. There is no need to compute the similarity score for these pairs during the second $APSS$. While executing the first $APSS$, the similarity score computed for some pairs would be less than 0.8. We can safely prune the similarity score computations of such pairs during the second $APSS$. Arguably, the more times $APSS$ is performed, the greater the opportunity to optimize the search by eliminating redundant calculations.

The $IAPSS$ problem should not be confused with other formulations of incremental problems. Incremental algorithms for various types of similarity searches have primarily addressed the challenge of handling perturbations in datasets themselves, when data records and/or their dimensions are added or removed[14]. Unlike these incremental methods, the IAPSS problem assumes that such datasets remained unchanged across different searches. Some incremental algorithms are designed to identify the *top-k* similar pairs [12]. But the $IAPSS$ problem requires all *matching pairs*. Incremental algorithms for the distance join [5] address problems similar to $IAPSS$ for distance measures, such

Nagiza F. Samatova is the corresponding author.

as the Euclidian distance. However, their techniques assume that the triangle inequality holds true for distance measures, which is not the case for similarity functions like the cosine similarity and the Tanimoto coefficient.

Given a dataset with n records in a d dimensional space where $d >> n$, a $na\ddot{i}ve$ algorithm for $IAPSS$ will compute and store the similarity scores between all pairs in $O(n^2 * d)$ time. However, this computational cost becomes prohibitively expensive for large-scale problems. To address this limitation our solution to the $IAPSS$ problem stores the computation history during each invocation of $IAPSS$ and later uses the history to systematically identify and effectively prune redundant computations. The compute and I/O intensive nature of the $IAPSS$ problem raises two key research challenges: (1) developing efficient techniques for I/O while using the computation history; and (2) efficiently identifying and pruning redundant computations. To address these challenges, we propose two major techniques: *history binning* and *index splitting*.

The history binning technique stores information about all pairs evaluated in the *current* invocation of $IAPSS$. Pairs are grouped based on their similarity scores and stored in binary files. This information is used in the *next* invocation of $IAPSS$ to avoid re-computation of known similarity scores. Grouping pairs enables our algorithm to read only the necessary parts of the computation history. The I/O for history binning is performed in parallel to the similarity score computation, which reduces the overhead in end-to-end execution time.

The index splitting technique divides the inverted index based on the values of t_{new} and t_{old}. This splitting enables our algorithm to avoid searching through a major part of the inverted index and to prune similarity score computations of pairs that exist in the computation history.

Lowering the value of the similarity threshold results in exploring a greater portion of the search space (i.e., the number of record pairs evaluated). The lowest similarity threshold value used in previous $IAPSS$ invocations defines the parts of the search space that have already been explored. Depending on the value of the current similarity threshold (t_{new}) and the previous lowest similarity threshold value (t_{old}), we identify three different cases for the $IAPSS$ problem: (1) *booting*, where the $IAPSS$ algorithm is executed for the first time on a given dataset, (2) *upscaling*, where $t_{old} \leq t_{new}$, and (3) *downscaling*, where $t_{old} > t_{new}$. The history binning technique is used in all three cases, while index splitting is required only for the downscaling case.

We incorporate both history binning and index splitting into the state-of-the-art $APSS$ algorithm [4], which enables us to split the $IAPSS$ computation into various independent subtasks that can be executed in parallel. This paper proposes the following contributions:

- Develops history binning and index splitting techniques that systematically identify and effectively prune redundant computations across multiple invocations of $APSS$.
- Incorporates our history binning and index splitting techniques into the state-of-the-art $APSS$ algorithm and parallelizes it, which leads to efficient end-to-end computation.
- Offers more responsive output than the state-of-the-art $APSS$ solution by almost instantaneously identifying pairs with high similarity scores. This responsive nature is particularly desirable for processing large datasets requiring multiple hours for complete execution.

TABLE I: Notations Used

table

Notation	Meaning
Given a dimension j	
$density(j)$	the number of vectors in V with non-zero projection along the dimension j
$global\_max\_weight[j]$	$x[j]$ such that $x[j] \geq y[j]$ for $\forall y \in V$
Given a vector x	
$x.max\_weight$	$x[k]$ such that $x[k] \geq x[i]$ for $1 \leq i \leq d$
$x.sum$	$\sum_{i=1}^{d} x[i]$
x'	the unindexed part of x
x''	the indexed part of x
$\|x\|$ (size of x)	the number of nonzero components in x
$\|\|x\|\|$ (magnitude of x)	$\sqrt{\sum_{i=1}^{d} x[i]^2}$
Given a pair of vectors (x, y)	
$dot(x, y)$	$\sum_{i} x[i] \cdot y[i]$
$cos(x, y)$	$dot(x, y)/(\|\|x\|\| \cdot \|\|y\|\|)$

We perform empirical studies using four real-world million record datasets derived from: (a) scientific literature collaboration in Medline [1] indexed papers, (b) Flickr[2] social networks, (c) LiveJournal[3] social networks, and (d) Orkut[4] social networks. We compare the performance of our algorithm against the state-of-the-art $APSS$ algorithm [4]. Depending on the similarity threshold variation, our speedups vary from $2X$ to over 10^5X.

II. Definitions and Notations

In this section we define the problem and other important terms referenced throughout the paper (please, see Table I for the summary of notations).

Definition 1 (*All Pairs Similarity Search*): The all pairs similarity search ($APSS$) problem is to find all pairs (x, y) and their exact value of similarity $sim(x, y)$ such that $x, y \in V$ and $sim(x, y) \; \geq t$, where

- V is a set of n real valued, non-negative, sparse vectors over a finite set of dimensions D and $|D| = d$;
- $sim(x, y) : V \times V \rightarrow [0, 1]$ is a symmetric similarity function; and
- t, $t \in [0, 1]$, is the similarity threshold.

[1] www.nlm.nih.gov/pubs/factsheets/medline.html
[2] www.flickr.com
[3] www.livejournal.com
[4] www.orkut.com

Definition 2 (*Incremental All Pairs Similarity Search*): The incremental all pairs similarity search problem is to the solve $APSS$ problem for a given similarity threshold value t_{new} when the $APSS$ problem is already solved for the least value of similarity threshold t_{old}.

Definition 3 (*Inverted Index*): The inverted index maps each dimension to a list of vectors with non-zero projections along that dimension. A set of all d lists $I = \{I_1, I_2, ..., I_d\}$, i.e., one for each dimension, represents the inverted index for V. Each entry in the list has a pair of values (x, w) such that if $(x, w) \in I_k$, then $x[k] = w$. The inverse of this statement is not necessarily true because some algorithms index only a part of each vector.

Definition 4 (*Candidate Vector* and *Candidate Pair*): Given a vector $x \in V$, any vector y in the inverted index is a candidate vector for x, if $\exists j$ such that $x[j] > 0$ and $(y, y[j]) \in I_j$. The corresponding pair (x, y) is a candidate pair.

Definition 5 (*Matching Vector* and *Matching Pair*): Given a vector $x \in V$ and the similarity threshold t, a candidate vector $y \in V$ is a matching vector for x if $sim(x, y) \geq t$. We say that y matches with x, and vice versa. The corresponding pair (x, y) is a matching pair.

During subsequent discussions we assume that all vectors are of unit length ($||x|| = ||y|| = 1$), and the similarity function is the cosine similarity. In this case, the cosine similarity equals the dot product, namely:

$$sim(x, y) = cos(x, y) = dot(x, y).$$

Our solution to the $IAPSS$ problem can be extended to other popular similarity measures like the Tanimoto coefficient and the Jaccard similarity using transformations presented by Bayardo *et al.* [4].

III. $APSS$ Algorithm

Because the proposed $IAPSS$ algorithm is based on the $APSS$ algorithm, here we briefly summarize $APSS$ and explain the All_Pairs algorithm [4], which is the state-of-the-art algorithm for $APSS$. The basic idea is similar to the way information retrieval systems answer queries [11]. Every vector in the dataset is considered to be a query and the corresponding matching pairs are found using the inverted index. Most of the time, however, the information retrieval system only requires the $top-k$ similar pairs, while $APSS$ requires all matching pairs.

The algorithm can be broadly divided into three phases: data preprocessing, pairs matching, and indexing. The preprocessing phase (lines 1-4, Algorithm 1) reorders vectors using a permutation Ω defined over V and components within each vector using permutation Π defined over D.

The matching phase (lines 6-14, Algorithm 1) finds candidate pairs and selects matching pairs from them. For a given vector $x \in V$, the $FindCandidates$ procedure scans the lists in the inverted index that correspond to the nonzero dimensions in x to find candidate pairs. Simultaneously, it accumulates a partial similarity score for each candidate pair. Some of the candidate pairs can be safely discarded by computing an upper bound on the similarity score in constant time. Otherwise, the exact similarity score is computed for the candidate pair.

The indexing phase adds a part of the given vector to the inverted index so that it can be matched with any of the remaining vectors (lines 15-21, Algorithm 1). The All_Pairs algorithm uses an upper bound on the possible similarity scores with only a part of the current vector (line 17, Algorithm 1). Once this bound reaches the similarity threshold, the remaining vector components are indexed. Please, refer to Bayardo *et al.* [9] for more details.

Algorithm 1: All_Pairs Algorithm.

```
   Input: $V$, $t$, $d$, $global\_max\_weight$, $\Omega$, $\Pi$
   Output: $MPS$ (Matching Pairs Set)
1  $MPS = \emptyset$;
2  $I_i = \emptyset$ ,$\forall$ $1 \leq i \leq d$;
3  $\Omega$ sorts vectors in decreasing order by $max\_weight$;
4  $\Pi$ sorts dimensions in decreasing order by density;
5  foreach $x \in V$ in the order defined by $\Omega$ do
6      $partScoreMap = \emptyset$;
7      FindCandidates($x, I, t, \Pi, partScoreMap$);
8      foreach $y$: $partScoreMap\{y\} > 0$ do
9          if $partScoreMap\{y\} + min(|y'|, |x|) *$
           $x.max\_weight * y'.max\_weight \geq t$ then
10             $s = partScoreMap\{y\} + dot(x, y')$;
11             if $s \geq t$ then
12                 $MPS = MPS \cup (x, y, s)$
13
14
15     $maxProduct = 0$;
16     foreach $i$: $x[i] > 0$, in the order defined by $\Pi$ do
17         $maxProduct = maxProduct + x[i] *$
           $min(global\_max\_weight[i], x.max\_weight)$;
18         if $maxProduct \geq t$ then
19             $I_i = I_i \cup \{x, x[i]\}$;
20             $x[i] = 0$;
21
22
23 return $MPS$
```

IV. $IAPSS$ Algorithm Overview

The $IAPSS$ algorithm is based on the observation that the proportion of the search space explored during the execution of a single $APSS$ invocation is inversely proportional to the value of the similarity threshold. If $t < t'$, then the search space explored while executing $APSS$ for t' is a subset of the search space explored for t. Therefore, the lowest previously used value of the similarity threshold is required while solving the $IAPSS$ problem. Depending on the relative values of the current similarity threshold (t_{new}) and the previous lowest similarity threshold value (t_{old}), Figure 1 gives an overview of the $IAPSS$ algorithm and there are three possible cases for the $IAPSS$ solution:

```
Procedure $FindCandidates$ procedure
  Input: $x$, $I$, $t$, $\Pi$, $partScoreMap$
  Output: modified $partScoreMap$, and $I$
  $remMaxScore = \sum_{i=1}^{d} x[i] * global\_max\_weight[i]$;
  $minSize = t/x.max\_weight$;
  foreach $i$: $x[i] > 0$, in the reverse order defined by $\Pi$ do
      Iteratively remove $(y, y[i])$ from front of $I_i$ while $|y| < minSize$;
      foreach $(y, y[i]) \in I_i$ do
          if $partScoreMap\{y\} > 0$ or $remMaxScore \geq t$ then
              $partScoreMap\{y\} = partScoreMap\{y\} + x[i] * y[i]$;

      $remMaxScore = remMaxScore - global\_maximum\_weight[i] * x[i]$;

```

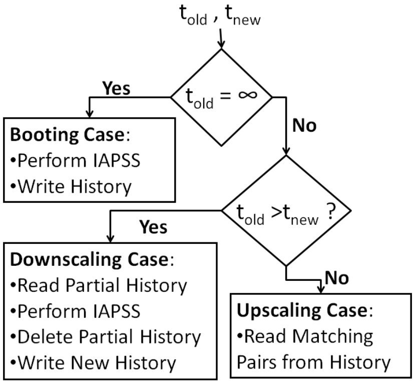


Fig. 1: $IAPSS$ Overview

figure

1) **Booting**: $t_{old} = \infty$, executing the $IAPSS$ algorithm for the first time on a given dataset.
2) **Upscaling**: $t_{old} \leq t_{new}$, reading a subset of pairs that are already present in the computation history.
3) **Downscaling**: $t_{old} > t_{new}$, potentially adding new similarity pairs to the computation history.

V. Booting

Booting is a relatively simple case of $IAPSS$ that performs $APSS$ while recording the computation history using history binning.

A. History Binning

Our $IAPSS$ algorithm takes a user defined parameter, P_{max}, that specifies the number of partitions for the similarity interval of $[0, 1]$. The interval is divided into equal sized non-overlapping P_{max} partitions. For example, if $P_{max} = 5$, then the similarity interval is divided into five partitions: $[0, 0.2)$; $[0.2, 0.4)$; $[0.4, 0.6)$; $[0.6, 0.8)$; and $[0.8, 1.0]$. Given a similarity value s, the corresponding partition number P_s can be calculated in constant time as $P_s = \lfloor s * P_{max} \rfloor$. For the special case of $s = 1$ the partition number is $P_{max} - 1$. All experiments reported in this paper are performed with $P_{max} = 20$. The effect of varying P_{max} is discussed in Section IX-C.

The history binning technique classifies candidate pairs into two types: *approximate pairs* and *exact pairs*. For each partition, pairs of each type are stored in different files, called *approximate pairs files* and *exact pairs files*, respectively. During the similarity score computation some candidate pairs are discarded after computing an upper bound on their similarity score because they do not satisfy the given threshold value (line 9, Algorithm 1). Such pairs are stored as approximate pairs in an *approximate pairs file* of the partition corresponding to the value of the upper bound on their similarity score. The exact similarity score is computed for the rest of the candidate pairs (line 10, Algorithm 1). These pairs are stored in an *exact pairs file* of the partition corresponding to their exact similarity score.

B. Booting Algorithm

Booting is the case of executing the $IAPSS$ algorithm for the first time on a given dataset. As there is no information available from any previous invocation of $APSS$, our $IAPSS$ algorithm simply uses the fastest algorithm for $APSS$ while storing the computation history. The booting algorithm is divided into two concurrent threads: the Candidate Pair Producer and the Candidate Pair Consumer. The Candidate Pair Producer executes the All_Pairs algorithm (please, refer to Algorithm 3), and the Candidate Pair Consumer writes candidate pairs to persistent storage (please, refer to Algorithm 4).

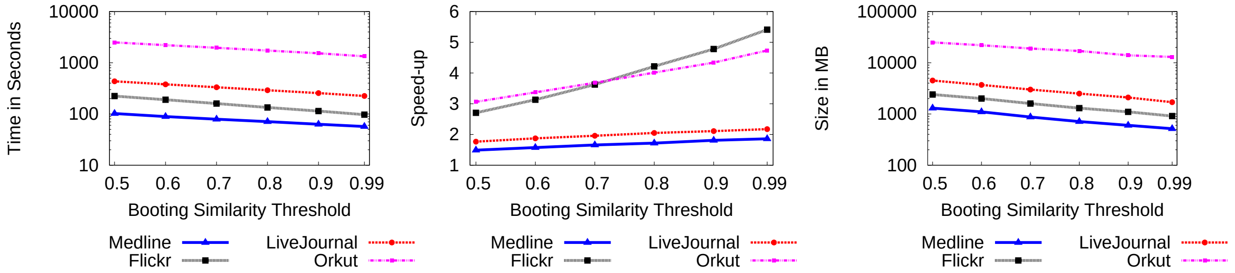
```
Algorithm 3: $Candidate\ Pair\ Producer$ Algorithm: Replace lines 9-13 of Algorithm 1 with the following pseudocode

$upperBound = partScoreMap\{y\} + min(sum(y') * x.max\_weight, sum(x) * y'.max\_weight)$;
if $upperBound \geq t$ then
    $s = partScoreMap\{y\} + dot(x, y')$;
    Add $(x, y, s, true)$ to $candidatePairQueue$;
    if $s \geq t$ then
        $MPS = MPS \cup (x, y, s)$

else
    Add $(x, y, upperBound, false)$ to $candidatePairQueue$;

```

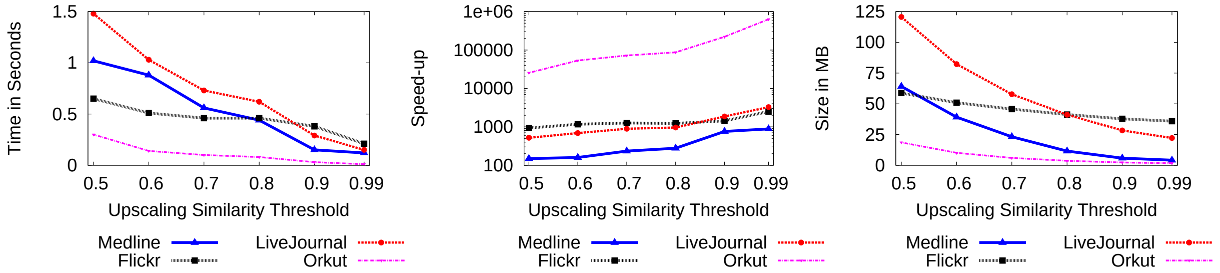
The producer and consumer share two data structures: the $doneFlag$ and $candidatePairQueue$. The $doneFlag$ is a binary variable that is initialized to false, and the Candidate pair producer sets it to $true$ when all candidate pairs are added to the $candidatePairQueue$. Each entry in the $candidatePairQueue$ has four components: the ids of



(a) Running Time of $IAPSS$ (b) Speed-up over All_Pairs (c) Size of Computation History

Fig. 2: Results for Booting

figure



(a) Running Time of $IAPSS$ (b) Speed-up over All_Pairs (c) Size of Computation History Read

Fig. 3: Results for Upscaling

figure

Algorithm 4: *Candidate Pair Consumer* Algorithm for a thread that writes candidate pairs to persistent storage

```
while $doneFlag$ not true do
    Dequeue all candidate pairs from $candidatePairsQueue$ in $writePairsSet$;
    foreach Element $w$ in $writePairsSet$ do
        $P_w = \lfloor w.score * P_{max} \rfloor$;
        if $w.isExact$ is true then
            Append entry $(w.x, w.y, w.score)$ to file for exact pairs corresponding to partition $P_w$
        else
            Append entry $(w.x, w.y)$ to file for approximate pairs corresponding to partition $P_w$
```

both vectors in the pair, the similarity score value, and a flag indicating if it is the exact score or an upper bound.

The producer thread performs the similarity computation and adds candidate pairs to the queue. The consumer thread removes candidate pairs from the queue and writes them to a file depending on the value of the similarity score. While writing approximate pairs, the value of the upper bound is discarded to reduce the size of data to be written. In later invocations of $IAPSS$, the value of the upper bound of an approximate pair can be computed using its partition number. However, it will be a loose upper bound.

The $IAPSS$ algorithm uses two tighter bounds on filtering conditions derived by Awekar and Samatova [3]. While searching for candidate pairs, the lower bound on the size of a candidate (line 2, $FindCandidates$ Procedure) is squared by the $IAPSS$ algorithm. While evaluating candidate pairs, the upper bound used by the $IAPSS$ algorithm on the similarity score is tighter (line 1, Algorithm 3) than the bound used by All_Pairs (line 9, Algorithm 1).

Figure 2a shows the running time of the $IAPSS$ booting algorithm for various similarity threshold values. Speed-up with respect to the All_Pairs algorithm is shown in Figure 2b. This speed-up is due to tighter bounds on the filtering conditions. Please, refer to Appendix for a description of the experimental set-up and datasets.

VI. Upscaling

Upscaling is another simple case of $IAPSS$, which only requires reading a part of the computation history and is the case where $t_{old} \leq t_{new}$. The set of matching pairs for threshold t_{new} will be a subset of the matching pairs for t_{old}. The matching pairs for t_{old} are a subset of all the candidate pairs for threshold t_{old} and have already been stored through history binning while executing $IAPSS$ for t_{old}. If a pair is a matching pair, then its similarity score is computed exactly (lines 3-7, Algorithm 3). Therefore, all matching pairs for threshold t_{old} have already been stored in exact pairs files. No separate search is required to find the matching pairs for threshold t_{new}.

Our algorithm only reads the computation history and outputs the matching pairs. It does not need to read the entire computation history because the history binning technique groups the pairs based on their similarity values. For current invocation of $IAPSS$, our algorithm first computes the partition number P_{new} corresponding to threshold t_{new}, and then reads the exact pairs files corresponding to all partitions P, $P_{new} \leq P < P_{max}$. The pairs satisfying the threshold t_{new} are then added to the output.

During our experiments, the first $IAPSS$ (booting) experiment used a threshold value of 0.5 and then performed upscaling with various similarity thresholds. For all datasets, upscaling was completed in less than two seconds (please, refer to Figure 3a); this is expected because the algorithm only reads and outputs matching pairs. It results in large speed-ups in the range 10^2X to 10^6X (please, refer to Figure 3b). The speed-up for the upscaling case is not dependent on the value t_{old} because the number of pairs read by the upscaling algorithm depends only on the value of t_{new}.

Grouping pairs by similarity score enables our algorithm to only read the required portions of the history. Figure 3c shows the effectiveness of grouping pairs using the history binning technique. Upscaling algorithms read at most five percent of the total history written during the booting case.

VII. Downscaling

Downscaling is the case of $t_{old} > t_{new}$. This is the trickiest case to handle because the search space explored for threshold t_{old} is a subset of the search space that needs to be explored for threshold t_{new}, and the challenge is to identify this overlap efficiently, which is achieved using history binning and index splitting.

A. Division of Search Space

The search space, that is, the set of candidate pairs C for the given similarity threshold t_{new} can be partitioned into two parts:

- C_{old} = The search space explored after running $IAPSS$ for threshold t_{old}, that is, the set of all candidate pairs present in the computation history; and
- $C_{new} = C - C_{old}$

C_{old} can be further partitioned into:

- C_{low} = Exact and approximate pairs having similarity score less than t_{new};
- C_{match} = Exact pairs having similarity scores greater than or equal to t_{new}; and
- C_{approx} = Approximate pairs having similarity score upper bounds greater than or equal to t_{new}.

Pairs in C_{low} can be ignored, as they will not satisfy threshold t_{new}. Pairs in C_{match} can be directly added to the output without re-computing the similarity score. These pairs have already been written in the exact pairs files. The similarity score must be recomputed for pairs in C_{approx}. The search space explored in the current execution of $IAPSS$ is limited to $C_{unknown} = C_{new} \cup C_{approx}$ and will result in pruning similarity score computations for pairs in $C_{known} = C - C_{unknown} = C_{low} \cup C_{match}$.

Algorithm 5: Downscaling Algorithm.

```
Input: $V$, $t$, $d$, $global\_max\_weight$, $\Omega$, $\Pi$, $P_{max}$
Output: $MPS$ (Matching Pairs Set)
$MPS = \emptyset$;
$I_i^{old} = \emptyset, \forall 1 \leq i \leq d$;
$I_i^{new} = \emptyset, \forall 1 \leq i \leq d$;
$\Omega$ sorts vectors in decreasing order by $max\_weight$;
$\Pi$ sorts dimensions in decreasing order by density;
foreach Partition $P : P_{new} \leq P < P_{max}$ do
    foreach Exact Pair $(x, y)$ in partition $P$ do
        if $s \geq t_{new}$ then
            $MPS = MPS \cup (x, y, s)$;
foreach $x \in V$ in the order defined by $\Omega$ do
    Initialize $approxList$ and $knownList$ to empty sets;
    foreach Partition $P : P_{new} \leq P < P_{max}$ do
        Add each $y$ to $ApproxList$, such that $(x, y)$ is an approximate pair in $P$;
        Delete $(x, y)$ from computation history;
    FindKnownCandidates();
    FindNewCandidates($x, I, t,$);
    foreach $y$: $partScoreMap\{y\} > 0$ do
        $upperBound = partScoreMap\{y\} + min(sum(y') * x.max\_weight, sum(x) * y'.max\_weight)$;
        if $upperBound \geq t$ then
            $s = partScoreMap\{y\} + dot(x, y')$;
            Add $(x, y, s, true)$ to $candidatePairQueue$;
            if $s \geq t$ then
                $MPS = MPS \cup (x, y, s)$
        else
            Add $(x, y, upperBound, false)$ to $candidatePairQueue$;
    SplitIndexVector();
$t_{old} = t_{new}$;
store updated value of $t_{old}$ to persistent storage;
return $MPS$
```

B. Index Splitting

The size of the inverted index is inversely proportional to the value of the similarity threshold (lines 16-21, Algorithm 1). The inverted index I^{old} is built for threshold value t_{old} and will be a subset of the inverted index I built for threshold value t_{new}. Our index splitting technique splits the

```
Procedure SplitIndexVector procedure
  Input: $x$, $I^{old}$, $I^{new}$, $t_{old}$, $t_{new}$, $\Pi$
  Output:
  $maxProduct \ = \ 0;$
  foreach $i$: $x[i] > 0$, in the order defined by $\Pi$ do
      $maxProduct = maxProduct + x[i] * min(global\_max\_weight[i], x.max\_weight);$
      if $maxProduct \geq t_{old}$ then
          $I_i^{old} \ = \ I_i^{old} \cup \{x, x[i]\};$
          $x[i] \ = \ 0;$
      else
          if $maxProduct \geq t_{new}$ then
              $I_i^{new} \ = \ I_i^{new} \cup \{x, x[i]\};$
              $x[i] \ = \ 0;$
```

inverted index I into the following two partitions: I^{old} and I^{new}, where $I^{new} = I \ - \ I^{old}$. Please refer to procedure $SplitIndexVector$ for details. Index splitting is used by the downscaling algorithm to partition the search space into C_{known} and $C_{unknown}$.

C. Downscaling Algorithm

The downscaling algorithm explores the $C_{unknown}$ search space and stores each evaluated pair in the computation history. The pairs in C_{match} and C_{approx} are read from computation history. C_{known} is found by traversing I^{old} and is used to prune redundant computations while finding and evaluating C_{new}. All pairs in $C_{unknown}$ are evaluated using the inverted index and added to the computation history. Old entries for the pairs in C_{approx} are removed from the computation history because their updated similarity scores will be stored during the current invocation of $IAPSS$.

1) Reading C_{match}: All pairs in C_{match} are already present in the computation history. They are read from the exact pairs files corresponding to each partition P, such that $P_{new} \leq P < P_{max}$ (lines 6-11, Algorithm 5). This step is similar to the upscaling case.

2) Reading and Evaluating C_{approx}: Similar to the pairs in C_{match}, pairs in C_{approx} can be read all at once from the approximate pairs files and evaluated directly. However, computing similarity scores directly for all these pairs will not be efficient, because computing the dot product requires serially traversing both vectors. Instead, we read the pairs in C_{approx} during the matching phase (lines 15-16, Algorithm 5). For a given vector x, the list of pairs in C_{approx} is stored in $approxList$. The partial similarity score for these pairs is calculated using the inverted index when finding C_{known} and C_{new} (please, refer to procedures $FindKnownCandidates$ and $FindNewCandidates$). The similarity score computation using the inverted index is more efficient than serially traversing the vectors. In addition, the evaluation for C_{approx} now piggybacks searching of C_{known} and C_{new}.

```
Procedure FindKnownCandidates procedure
  Input: $x$, $I^{old}$, $t_{old}$, $\Pi$, $partScoreMap$, $knownList$, $approxList$
  Output: modified $partScoreMap$, and $knownList$
  $partScoreMap \ = \ \emptyset;$
  $remMaxScore \ = \sum_{i=1}^{d} x[i] * global\_max\_weight[i];$
  $minSize_{old} \ = \ (t_{old}/x.max\_weight)^2;$
  foreach $i$: $x[i] > 0$, in the reverse order defined by $\Pi$ do
      Iteratively ignore $(y, y[i])$ from front of $I_i^{old}$ while $|y| < minSize_{old}$;
      foreach $(y, y[i]) \in I_i^{old}$ do
          if $y \in approxList$ then
              $partScoreMap\{y\} \ = \ partScoreMap\{y\} \ + \ x[i] * y[i];$
          else
              Add $y$ to $knownList$;
      $remMaxScore \ = \ remMaxScore \ - \ global\_maximum\_weight[i] * x[i];$
      if $remMaxScore < t_{old}$ then
          return
```

3) Finding C_{known}: Finding all the pairs in C_{known} can be accomplished by reading the entire computation history. However, finding C_{known} from the inverted index is more efficient because it is an in-memory data structure. For a given vector x, the $FindKnownCandidates$ procedure finds pairs in C_{known}. It traverses the inverted index in the same manner as the $FindCandidates$ procedure in Algorithm 1. However, the similarity score is computed only for pairs in the $approxList$. The list of pairs in C_{known} is stored in the $knownList$.

4) Finding C_{new}: For a given vector x, the $FindNewCandidates$ procedure finds candidate vectors in C_{new}. The procedure is similar to the $FindCandidates$ procedure in Algorithm 1. However, it does not search the part of the index that was traversed by $FindKnownCandidates$. If any candidate vector y is present in that part of the index, then by definition $(x, y) \ \in \ C_{old}$. Therefore, any pair in C_{new} cannot be present in that part of the index. Simultaneously, the partial similarity score is accumulated in $partScoreMap$ for all pairs in $C_{unknown}$.

5) Evaluating and Storing $C_{unknown}$: The partial similarity score of all the candidate pairs in $C_{unknown}$ is stored in $partScoreMap$. These candidate pairs are evaluated and stored exactly like the booting case (lines 19-29, Algorithm 5).

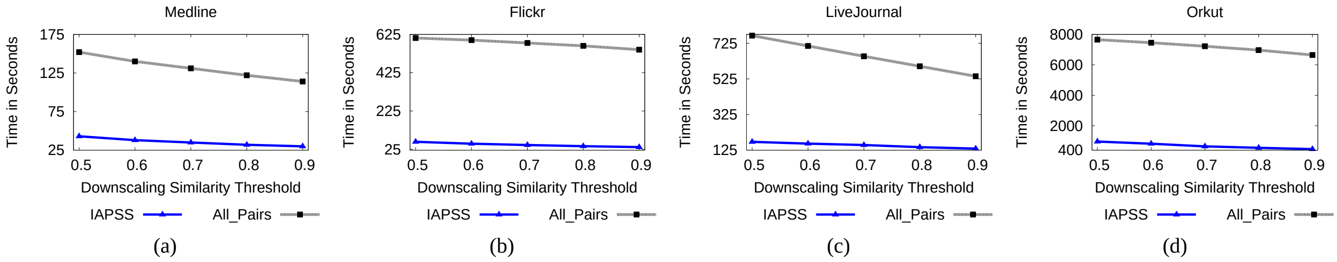


Fig. 5: Comparison of Running Time for Downscaling with All_Pairs

figure

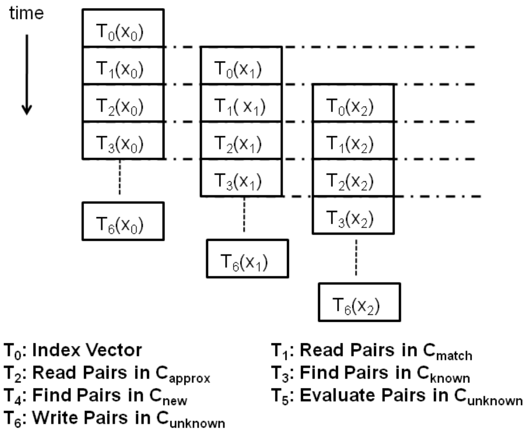


Fig. 4: Overview of Parallelization

figure

VIII. Parallelization

Additional performance gains may be attained by interleaving I/O and computation, and by concurrently executing various subtasks, such as finding C_{new}, C_{known}, and evaluating $C_{unknown}$. Out of the three cases for the $IAPSS$ problem, the solution for the upscaling case only consists of reading matching pairs from the exact pairs files, and does not require parallelization. The solution for the booting case uses parallelization to multiplex I/O with the computation. The same is true in the solution presented in Algorithm 5. However, various smaller subtasks presented in Section VII-C present opportunities for parallelizing the downscaling computation. These subtasks can run in parallel, while data flows through these subtasks.

Figure 4 shows the parallelization outline. It works as a pipeline of producers and consumers. Each task works as a producer for its successor, and works as a consumer for its predecessor. For example, the task T_4 finds the set of pairs in C_{new} for a given vector x, and adds it to the queue shared with task task T_5. The vector x and the corresponding pairs in C_{new} are then removed from the queue by the task T_5. In our implementations, each task runs as a thread and synchronizes with its neighbors using shared-memory data structures. Data flows from top to bottom in this pipeline. Synchronization between the last two tasks, T_5 and T_6, was presented in Algorithms 3 and 4. For other producer-consumer pairs, synchronization scheme is similar.

Figure 5 shows running time comparisons for the $IAPSS$ downscaling case and the All_Pairs algorithm. We started with a booting similarity threshold of 0.99. Then we reduced the similarity threshold to 0.5 in 0.1 decrement steps. The end-to-end running time is the most important measure for comparing the $IAPSS$ solution to the All_Pairs algorithm. The results for other comparison factors are available on the Web [1], such as the size of the search space and the amount of I/O performed.

IX. End-To-End $IAPSS$ Performance

In this section, we present results for experiments that are relevant across all three cases of the $IAPSS$ algorithm using three metrics: (1) query responsiveness, (2) speed-up, and (3) sensitivity. We chose the following set of similarity threshold values for the experiments:

$$T = \{0.99, 0.9, 0.8, 0.7, 0.6.0.5\}.$$

A. Query Responsiveness to Similarity Value Changes in IAPSS

An algorithm has high query responsiveness if it immediately generates the majority of its output and then computes the remaining portion of the output. Other algorithms that use the $IAPSS$ output can benefit from the algorithm's query responsiveness. These algorithms do not need to wait until all matching pairs are found. Instead, they can start using the matching pairs as they are identified. This is particularly useful while processing large datasets, where the total running time for finding all matching pairs may take hours.

The query responsiveness of the $IAPSS$ solution for the booting case, is similar to the All_Pairs algorithm. The $IAPSS$ solution directly outputs all matching pairs by reading them from the computation history for the upscaling case. For the downscaling case, the $IAPSS$ algorithm immediately outputs pairs in C_{match} and then finds pairs in $C_{unknown}$. All pairs having similarity scores greater than or equal to t_{old} are present in C_{match}, i.e., pairs with high similarity value are immediately identified by the $IAPSS$ solution. Figure 6 shows the ratio of the number of pairs in C_{match} to the total number of matching pairs for various

```
Procedure $FindNewCandidates$ procedure
  Input: $x$, $I^{old}$, $t_{old}$, $\Pi$, $partScoreMap$, $knownList$,
         $approxList$
  Output: modified $partScoreMap$
  $remMaxScore = \sum_{i=1}^{d} x[i] * global\_max\_weight[i]$;
  $minSize_{old} = (t_{old}/x.max\_weight)^2$;
  $minSize_{new} = (t_{new}/x.max\_weight)^2$;
  foreach $i$: $x[i] > 0$, in the reverse order defined by $\Pi$
  do
    Iteratively remove $(y, y[i])$ from front of $I_i^{new}$, and
    $I_i^{old}$ while $|y| < minSize_{new}$;
    if $remMaxScore \geq t_{old}$ then
      foreach $(y, y[i]) \in I_i^{old}$ while $|y| < minSize_{old}$
          /* remaining part in the list
          was traversed by
          $FindKnownCandidates$ procedure
          */
      do
        if $y \notin knownList$ then
          $partScoreMap\{y\} =$
          $partScoreMap\{y\} + x[i] * y[i]$;

      foreach $(y, y[i]) \in I_i^{new}$ do
        if $y \notin knownList$ then
          $partScoreMap\{y\} =$
          $partScoreMap\{y\} + x[i] * y[i]$;

    else
      foreach $(y, y[i]) \in I_i^{new} \cup I_i^{old}$ do
        if $y \notin knownList$ then
          if $partScoreMap\{y\} > 0$ or
          $remMaxScore \geq t_{new}$ then
            $partScoreMap\{y\} =$
            $partScoreMap\{y\} + x[i] * y[i]$;

    $remMaxScore = remMaxScore -$
    $global\_maximum\_weight[i] * x[i]$;
```

downscaling similarity threshold values. This ratio represents the part of the output immediately generated by the downscaling algorithm.

B. Extreme Cases Speed-up

The speed-up achieved by the $IAPSS$ algorithm depends on how the similarity threshold is varied. If the $IAPSS$ algorithm is executed n times over a given dataset, then the following are the best and worst cases for the end-to-end running time.

Best Case: Execute booting followed by $(n-1)$ upscaling cases.

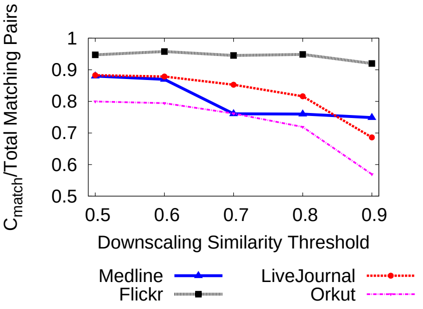


Fig. 6: Fraction of Matching Pairs Immediately Found by Downscaling Algorithm
figure

Worst Case: Execute booting followed by $(n-1)$ downscaling cases.

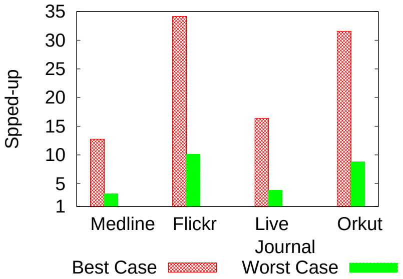


Fig. 7: Best and Worst Case Speed-up for Similarity Values in Set T
figure

The best case is obtained by sorting the threshold values in the threshold set T in increasing order and then executing $IAPSS$. The worst case is obtained by sorting the threshold values in decreasing order and then executing $IAPSS$. Figure 7 shows the best and worst case speed-ups achieved by the $IAPSS$ solution compared to the All_Pairs algorithm. The speed-up is computed by comparing the total running time over all similarity threshold values in the set T. If the value of $|T|$ is increased, i.e., if $IAPSS$ is executed more often on the same dataset, then the resultant speed-up will increase because the $IAPSS$ algorithm will prune more redundant computations from later invocations.

The external algorithm that invokes the $IAPSS$ algorithm can implement various strategies to achieve the best case speed-up. A particular lowest similarity threshold can be predicted for some applications based on historical data and empirical knowledge. Alternatively, the external algorithm can also buffer the $IAPSS$ request for some time instead of executing it immediately. Depending on the nature of the application, it can wait for a certain time to check if any other $IAPSS$ requests have been received with lower similarity threshold values.

C. Sensitivity to Varying P_{max}

The P_{max} parameter is used to divide the similarity range into equal sized partitions. For a given value of t_{new}, the

$IAPSS$ algorithm has to read the computation history for all partitions $P : P_{new} \leq P < P_{max}$. Some pairs in the partition P_{new} will not satisfy the similarity threshold, but must be read anyway. This overhead is attenuated if the partition floor equals t_{new}, i.e., $P_{new} = t_{new} * P_{max}$. However, we observed that this overhead is not significant. During our experiments, we varied the P_{max} parameter from 3 to 25. The variation in total running time for the best case and the worst case for values in T was less than ten percent.

X. Conclusions and Future Work

The Incremental All Pairs Similarity Search ($IAPSS$) problem is introduced and a solution is proposed. The major features of the solution are the following:

- Redundant computations in response to varying similarity thresholds across multiple invocations of $APSS$ on the same dataset are systematically identified and effectively pruned using the proposed history binning and index splitting techniques.
- Additional performance gains are attained by parallelizing our $IAPSS$ algorithm to take advantage of modern multi-core processors.
- Query responsiveness is improved for our $IAPSS$ solution, compared to the All_Pairs $APSS$ algorithm, because it almost instantaneously output pairs with high similarity values.

The compounded effect of these approaches resulted in speed-ups of $2X$ to over 10^5X on four large-scale real-world datasets.

Our current parallel solution for $IAPSS$ is limited to a shared-memory multi-core system. Scaling the $IAPSS$ solution using both shared and distributed memory systems is an interesting direction for future work and may enable even larger datasets to be processed in the future.

Appendix

We empirically evaluate the effectiveness of our techniques by performing experiments on four real-world datasets for both the cosine similarity and the Tanimoto coefficient. Results for both similarity measures are quite similar. In this paper, we only present results for the cosine similarity for the sake of brevity. More details about the results for the Tanimoto coefficient can be downloaded from the Web [1].

All our implementations are in C++ and we used the standard template library for most of the data structures. We used the $dense\_hash\_map\_class$ [5] from Google^{TM} for the hash based partial score accumulation. We used the GNU gcc 4.1.2 compiler and the $-O3$ option for optimization. We used the $pthreads$ library for multithreading to implement parallelization. All experiments were performed on a 2.6 GHz Intel^{TM} Xeon^{TM} class machine with eight CPU cores and 16 GB of main memory. The code and datasets are available for download on the Web [1].

TABLE II: Data Sets Used

table

Data Set	$n = d$	Total Non-zero Components	Average Size
Medline	1565145	18722422	11.96
Flickr	1441433	22613976	15.68
LiveJournal	4598703	77402652	16.83
Orkut	2997376	223534153	74.57

A. Datasets

One of the datasets comes from the scientific literature collaboration information in Medline indexed papers, while the rest come from popular online social networks: Flickr, LiveJournal and Orkut. These datasets represent a variety of large-scale web-based applications like digital libraries and online social networks that we are primarily interested in.

The distribution of the vector sizes in these datasets is the power law distribution [7], [3], [4]. These datasets are high dimensional and sparse (please, refer to Table II). The ratio of the average vector size to the total number of dimensions is less than 10^{-4}. All these characteristics are common across datasets generated and used by many large-scale web based applications [13], [4]. These applications have to solve the $IAPSS$ problem for high-dimensional datasets with millions of records, which are often sparse. Therefore, we expect our history binning and index splitting techniques to be relevant to other similar datasets as well.

1) Medline: This dataset was selected to investigate possible applications for large web-based scientific digital libraries like PubMed, the ACM Digital Library, and CiteSeer. We use the dataset prepared by the Auton Lab of Carnegie Mellon University. We are interested in finding pairs of authors with similar collaboration patterns. Each vector represents the collaboration pattern of an author over the space of all authors. Two authors are considered collaborators if they write at least two papers together. Similar strategies were used in previous work [4] to eliminate accidental collaborations. We use the weighting scheme of Newman [8] to derive the collaboration weight between any two authors. If k authors have co-authored a paper, then it adds $1/(k-1)$ to the collaboration weight of each possible pair of authors of that paper. All vectors are then normalized to unit-length.

2) Flickr, LiveJournal and Orkut: These three datasets were selected to explore potential applications for large online social networks. We are interested in finding user pairs with similar social networking patterns. Such pairs are used to generate more effective recommendations based on collaborative filtering [10]. We use the dataset prepared by Mislove *et al.* [7]. Every user in the social network is represented by a vector over the space of all users. A user's vector has a non-zero projection along those dimensions that correspond to other users in his/her friend list. However, the weights of these social network links are unknown. Therefore, we applied the weight distribution from the Medline dataset. To ensure that our results are not specific only to the selected weight distribution, we also conducted experiments by generating the weights randomly. The results were similar

[5] code.google.com/p/google-sparsehash/

and are available on the Web [1].

Acknowledgment

This work is performed as part of the Scientific Data Management Center (http://sdmcenter.lbl.gov) under the Department of Energy's Scientific Discovery through Advanced Computing program (http://www.scidac.org). Oak Ridge National Laboratory is managed by UT-Battelle for the LLC U.S. D.O.E. under contract no. DEAC05-00OR22725.

References

[1] Code and data sets for our algorithms : `www4.ncsu.edu/~acawekar/snakdd/`.

[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB '06*.

[3] A. Awekar and N. F. Samatova. Fast matching for all pairs similarity search. Technical Report TR-2009-14, CSC Department, NC State University, May 2009.

[4] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW '07*.

[5] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD '98*.

[6] R. Jarvis and E. Patrick. Clustering using a similarity measure based on shared near neighbors. *Computers, IEEE Transactions on*, C-22(11), Nov. 1973.

[7] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *IMC '07*.

[8] M. E. J. Newman. Scientific collaboration networks. ii. shortest paths, weighted networks, and centrality. *Physical Review*, 64(016132), 2001.

[9] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD '04*.

[10] E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating similarity measures: a large-scale study in the orkut social network. In *KDD '05*.

[11] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Process. Manage.*, 31(6):831–850, 1995.

[12] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE '09*.

[13] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW '08*.

[14] D. Zhou, S. Zhu, K. Yu, X. Song, B. L. Tseng, H. Zha, and C. L. Giles. Learning multiple graphs for document recommendations. In *WWW '08*.