Incremental All Pairs Similarity Search for Varying Similarity Thresholds with Reduced I/O Overhead

Amit Awekar, Nagiza F. Samatova1, and Paul Breimyer
acawekar@ncsu.edu, samatovan@ornl.gov, pwbreimy@ncsu.edu
North Carolina State University, Raleigh, NC
Oak Ridge National Laboratory, Oak Ridge, TN
1Corresponding Author

Abstract—All Pairs Similarity Search (APSS) is the problem of finding all pairs of records with similarity scores above a specified threshold. Incremental All Pairs Similarity Search (IAPSS) is the problem of performing APSS multiple times over the same dataset by varying the similarity threshold. This problem is ubiquitous in many real-world systems like search engines, online social networks, and digital libraries. A significant part of the computation is redundant across multiple invocations of APSS. Our solution to the IAPSS problem avoids these redundant computations by storing the history of previous APSS invocations and splitting the inverted index that maps each dimension into a list of records that have non-zero projections along that dimension. The size of the computation history increases quadratically with the number of records in the dataset. We introduce the concept of a similarity floor to store partial contributions, reducing the size of the computation history by at least an order of magnitude. We empirically evaluate the effectiveness of our techniques using four real-world large-scale datasets. Our IAPSS solution achieves speed-ups in the order of 2X to over 105 X over the state-of-the-art APSS algorithm, while reducing the size of the computation history by at least an order of magnitude.

Keywords: inverted index, similarity search, data mining

1. Introduction

Many real-world systems, such as search engines, online social networks and digital libraries frequently search for all pairs of records with similarity above a specified threshold [5], [13]. Similarity between two records is defined via some similarity measure, such as the cosine similarity or the Tanimoto coefficient. In the literature, this problem is referred to as similarity join [13] or all pairs similarity search (APSS) [5]. For example, the Jarvis-Patrick clustering algorithm sparsifies the similarity score matrix by retaining only those entries that satisfy a predefined threshold [7].

Meaningful similarity threshold values are data dependent. Domain experts often employ a trial and error approach by looking at the quality of output as a means to determine the threshold value. 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.

Performing APSS multiple times on the same dataset by varying the similarity threshold value leads to another important problem that we refer to as the incremental all pairs similarity search (IAPSS). The IAPSS problem is challenging to solve when it is applied frequently or over large datasets. For example, to detect near duplicate documents [13], a news search engine has to solve the IAPSS problem every few minutes over a small subset of the web, whereas a web search engine has to solve the IAPSS problem once every few days, but over the entire web.

A significant part of the computation is redundant across multiple invocations of APSS. Existing solutions for APSS [13], [5], [2], [10] do not exploit this fact. Instead, each APSS instance 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. The similarity score computations of such pairs can be safely pruned during the second APSS.

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 [6] address problems similar to IAPSS for distance measures, such 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 naïve algorithm for IAPSS will compute and store the similarity scores between all pairs in \( O(n^2 \cdot d) \) time. However, this computational cost becomes prohibitively expensive for large-scale problems. The compute and I/O intensive nature of the IAPSS problem raises two key research challenges: (1) developing efficient I/O techniques to deal with possibly large history data; 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 selectively stores information about 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.

The history binning technique requires a user defined parameter similarity floor \((\text{simFloor})\). Any pair having a similarity score below \(\text{simFloor}\) is not stored in the computation history. This selective storing reduces the I/O overhead because a significant portion of pairs have very low similarity score and are rarely required. If such ignored pairs are required in the next invocation of IAPSS then their similarity score is recomputed.

The index splitting technique divides the inverted index based on the values of \(t_{\text{new}}\) and \(t_{\text{old}}\). 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.

A greater portion of the search space (i.e., the number of record pairs evaluated) is explored if the similarity threshold is lowered. The lowest similarity threshold value used in previous IAPSS invocations determines the parts of the search space that have already been explored. Depending on the value of the current similarity threshold \((t_{\text{new}})\), the previous lowest similarity threshold value \((t_{\text{old}})\), and the value of similarity floor \((\text{simFloor})\), we identify four 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 \(t_{\text{old}} \leq t_{\text{new}}\), (3) downscaling, where \(t_{\text{old}} > t_{\text{new}}\), and (4) flooring, where \(t_{\text{new}} < \text{simFloor}\). The history binning technique is used in all four 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 [5], which enables the IAPSS computation to be split into various independent subtasks that can be executed in parallel. This paper contributes to the research on APSS in following ways:

- Develops history binning and index splitting techniques that systematically identify and effectively prune redundant computations across multiple APSS invocations.
- Incorporates our history binning and index splitting techniques into the state-of-the-art APSS algorithm while reducing I/O overhead.

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

2. Definitions and Notations

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

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

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

**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 \(x[k] = w\).
Definition 4 (Candidate Vector and Candidate Pair): Given a vector \( x \in V \), any vector \( y \) in the inverted index is a candidate vector, 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 \( \text{sim}(x, y) \geq t \). 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:

\[
\text{sim}(x, y) = \cos(x, y) = \text{dot}(x, y).
\]

### 3. APSS Algorithm

Because the proposed IAPSS algorithm is based on the APSS algorithm, here we briefly summarize APSS and explain the All_Pairs algorithm [5], which is the state-of-the-art algorithm for APSS. The basic idea is similar to the way information retrieval systems answer queries. For every vector in the dataset considered as a query, the corresponding matching pairs are found using the inverted index.

The algorithm can be broadly divided into three phases:

**Preprocessing** (lines 1-4, Algorithm 1) reorders vectors using a permutaion \( \Omega \) defined over \( V \) and components within each vector using permutation \( \Pi \) 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 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.

### 4. 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 \( t_{\text{new}} \), the previous lowest similarity threshold value \( t_{\text{old}} \), and similarity floor \( \text{simFloor} \), Figure 1 gives an overview of the IAPSS algorithm. There are four possible cases for the IAPSS solution:

1. **Booting**: \( t_{\text{old}} = \infty \), executing the IAPSS algorithm for the first time on a given dataset.
2. **Upscaling**: \( t_{\text{old}} \leq t_{\text{new}} \), reading a subset of pairs that are already present in the computation history.
3. **Downscaling**: \( t_{\text{old}} > t_{\text{new}} \), potentially adding new similarity pairs to the computation history.
4. **Flooring**: \( t_{\text{new}} < \text{simFloor} \), deleting all computation history and performing IAPSS like the booting case.

### 5. Booting

#### 5.1 History Binning

Our IAPSS algorithm takes a user defined parameter, \( P_{\text{max}} \), that specifies the number of partitions for the similarity interval of \([0, 1]\). The interval is divided into equal sized non-overlapping \( P_{\text{max}} \) partitions. Given a similarity value \( s \),

```
Algorithm 1: All_Pairs Algorithm.

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

Procedure FindCandidates procedure
1: Procedure FindCandidates procedure
2: procedure: FindCandidates procedure
3: Input: x, I, t, II, partScoreMap
4: Output: modified partScoreMap, and I
5: \( \text{remMaxScore} = \sum_{i=1}^{d} x[i] \cdot \text{global\_max\_weight}[i]; \)
6: \( \text{minSize} = t / x, \text{max\_weight}; \)
7: foreach i: \( x[i] > 0 \), in the reverse order defined by II do
8: Iteratively remove \((y, y[i])\) from front of \( I_i \) while \( |y| < \text{minSize}; \)
9: foreach \((y, y[i]) \in I_i \) do
10: if partScoreMap\{y\} > 0 or \( \text{remMaxScore} \geq t \) then
11: \( \text{partScoreMap}\{y\} = \text{partScoreMap}\{y\} + x[i] \cdot y[i]; \)
12: \( \text{remMaxScore} = \text{remMaxScore} - \text{global\_max\_weight}[i] \cdot x[i]; \)
13: 

![IAPSS Overview](image)

Fig. 1: IAPSS Overview

the corresponding partition number \( P_s \) can be calculated in constant time as \( P_s = [s \cdot P_{\text{max}}]. \) For the special case of \( s = 1 \) the partition number is \( P_{\text{max}} - 1. \) All experiments reported in this paper are performed with \( P_{\text{max}} = 20. \) During our experiments, the effect of varying \( P_{\text{max}} \) in the range 3 to 25 was negligible.

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 ignored if the upper bound on their similarity score does 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.

Our IAPSS algorithm takes another user defined parameter \( \text{simFloor} \) which determines the cut-off value of similarity score for storing a pair in the computation history. Any pair having similarity score less than \( \text{simFloor} \) is not stored in the computation history (lines 6 and 10, Algorithm 3). We observe that out of possible \( O(n^2) \) pairs, most pairs have low similarity score. During our experiments we observe that I/O overheads are reduced by at least an order of magnitude by eliminating such pairs from the computation history (please, refer to Figure 2). For the booting case, the value of \( \text{simFloor} \) has to be less than or equal to \( t_{\text{new}}. \) All pairs with similarity score at least \( t_{\text{new}} \) must be present in the computation history for upscaling and downscaling.

5.2 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 producer executes the All_Pairs algorithm (please, refer to Algorithm 3), and the consumer writes candidate pairs to persistent storage (please, refer to Algorithm 4).

Algorithm 3: Candidate Pair Producer Algorithm: Following pseudocode replaces lines 9-13 of Algorithm 1

1. \( upperBound = \text{partScoreMap}\{y\} + \text{min} \{(\text{sum}(y') \ast x, \text{max\_weight}, \text{sum}(x') \ast y', \text{max\_weight})\}; \)
2. if \( upperBound \geq t \) then
3. \( s = \text{partScoreMap}\{y\} + \text{dot}(x, y'); \)
4. if \( s \geq t \) then
5. \( \text{MPS} = \text{MPS} \cup (x, y, s) \)
6. if \( s \geq \text{simFloor} \) then
7. \( \text{Add} (x, y, s, \text{true}) \) to candidatePairQueue;
8. else
9. if \( upperBound \geq \text{simFloor} \) then
10. \( \text{Add} (x, y, upperBound, \text{false}) \) to candidatePairQueue;
11. 

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
Algorithm 4: Candidate Pair Consumer Algorithm that writes candidate pairs to persistent storage

```plaintext
while doneFlag not true do
  Dequeue all candidate pairs from candidatePairsQueue in writePairsSet;
  foreach Element w in writePairsSet do
    \( P_w = \lfloor w.score \times P_{\text{max}} \rfloor \);
    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 \( P_w \)
  end for
end while
```

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 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 performs the similarity computation and adds candidate pairs to the queue. The consumer removes candidate pairs from the queue and writes them to a file depending on the similarity score. While writing approximate pairs, the upper bound is discarded to reduce the size of data to be written. In later invocations of IAPSS, 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).

Speed-up with respect to the All_Pairs algorithm is shown in Figure 3a. This speed-up is due to tighter bounds on the filtering conditions. Please, refer to the Appendix for description of the experimental set-up and datasets.

### 6. Upscaling

Upscaling only requires reading a part of the computation history and is the case where \( t_{\text{old}} \leq t_{\text{new}} \). The set of matching pairs for threshold \( t_{\text{new}} \) will be a subset of the matching pairs for \( t_{\text{old}} \). The matching pairs for \( t_{\text{old}} \) are a subset of all the candidate pairs for threshold \( t_{\text{old}} \) and have already been stored through history binning while executing IAPSS for \( t_{\text{old}} \). If a pair is a matching pair, then its similarity score is computed exactly (lines 3-7, Algorithm 3).
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 \( P \) corresponding to threshold \( t_{new} \), and then reads the exact pairs files corresponding to all partitions. For all datasets, upscaling was completed in less than two seconds because the algorithm only reads and extracts the computation history. Our index splitting technique splits the inverted index into the following two partitions: \( \text{new} \) and \( \text{old} \).

During our experiments, the first IAPSS (booting) experiment used a booting threshold and similarity floor value of 0.5, and then performed upscaling with various similarity thresholds. For all datasets, upscaling was completed in less than two seconds because the algorithm only reads and outputs matching pairs. It results in large speed-ups in the range \( 10^2 \times \) to \( 10^5 \times \) (please, refer to Figure 3b). The speed-up for the upscaling case is not dependent on the values of \( t_{old} \) or \( t_{sim} \) because the number of pairs read by the upscaling algorithm depends only on the value of \( t_{new} \). Grouping pairs by similarity floor enables our algorithm to only read the required portions of the computation history.

7. Downscaling

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

- \( C_{old} = \) 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 already exist 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 \( C_{known} = C - C_{unknown} = C_{low} \cup C_{match} \).

7.1 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 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 \( C_{unknown} \).

```
Procedure SplitIndexVector procedure

Input: \( x, t_{old}, t_{new}, t_{old}, t_{new}, \Pi \)
Output:
maxProduct = 0;
foreach \( i: x[i] > 0 \), in the order defined by \( \Pi \)
    maxProduct = maxProduct + \( x[i] \) * \( \min(\text{global max weight}[i], x, \text{max weight}) \);
if \( \maxProduct \geq t_{old} \) then
    \( I_{old} = I_{old} \cup \{x, x[i]\} \);
    \( x[i] = 0 \);
else
    if \( \maxProduct \geq t_{new} \) then
        \( I_{new} = I_{new} \cup \{x, x[i]\} \);
    \( x[i] = 0 \);  
```

7.2 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. Due to space restrictions, we forgo the details of the downscaling algorithm. Please refer to Awekar et al. [4].

Figure 3c shows the speed-up achieved by our IAPSS algorithm over the All_Pairs algorithm for the downscaling case. We started with the booting similarity threshold of 0.99 and similarity floor value of 0.5. Then we reduced the similarity threshold to 0.5 in 0.1 decrement steps.

8. Flooring

The flooring case is similar to the downscaling case with the exception that \( t_{sim} > t_{new} \). However, the solution for the flooring case is similar to the booting case.

The computation history is in an inconsistent state for the flooring case. As \( t_{sim} > t_{new} \), some pairs in \( C_{match} \) and \( C_{approx} \) will have similarity score less than \( t_{sim} \). Such pairs are not present in the computation history. Therefore, the downscaling solution cannot be applied to the flooring case. The pairs missing from the computation
history can be found only by performing the IAPSS computation from scratch. The inconsistent computation history is deleted, followed by executing the booting case.

The IAPSS algorithm performance for the flooring case is similar to the booting case. Time spent in deleting inconsistent computation history is negligible compared to the end-to-end running time of the IAPSS algorithm.

9. 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.

Best Case: Booting followed by \((n - 1)\) upscalings.
Worst Case: Booting followed by \((n - 1)\) floorings.

![Fig. 4: Best and Worst Case Speed-up for Values in Set \( T \)](image)

We chose the following set of similarity threshold values for the experiments: \( T = \{0.99, 0.9, 0.8, 0.7, 0.6, 0.5\} \). 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, then the resultant speed-up will increase.

10. 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 IAPSS are systematically identified and effectively pruned using the proposed history binning and index splitting techniques.
- Selectively storing the computation history reduced I/O overhead significantly.

The compounded effect of these approaches resulted in speed-ups of \( 2^X \) to over \( 10^5 \cdot X \) on four large-scale datasets.

To process even larger datasets in the future, scaling the IAPSS solution using both shared and distributed memory systems is an interesting direction for future work.

Acknowledgment

This work is performed as part of the Scientific Data Management Center (http://sdmccenter.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.

Appendix

Please, refer to Table 2 for information about the datasets used. More details about these data sets are available in [4]. We performed experiments for both the cosine similarity and the Tanimoto coefficient. Results for both similarity measures are quite similar. All experiments were performed on a 2.6 GHz Intel \( \text{Xeon}^\text{TM} \) class machine with eight CPU cores and 16 GB of main memory. The code, datasets and additional details about experiments are available for download on the Web [1].

Table 2: Data Sets Used

<table>
<thead>
<tr>
<th>Data Set</th>
<th>( n = d ) Size</th>
<th>Total Non-zero Components</th>
<th>Average Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medline</td>
<td>1565145</td>
<td>18722322</td>
<td>11.96</td>
</tr>
<tr>
<td>Flickr</td>
<td>1441433</td>
<td>22613976</td>
<td>15.68</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>3598705</td>
<td>77402652</td>
<td>16.83</td>
</tr>
<tr>
<td>Orkut</td>
<td>2997376</td>
<td>223534153</td>
<td>74.57</td>
</tr>
</tbody>
</table>

References