ABSTRACT

AWEKAR, AMIT C. Fast, Incremental, and Scalable All Pairs Similarity Search. (Under the direction of Professor Nagiza F. Samatova and Professor Anatoli V. Melechko).

Searching pairs of similar data records is an operation required for many data mining techniques like clustering and collaborative filtering. With the emergence of the Web, scale of the data has increased to several millions or billions of records. Business and scientific applications like search engines, digital libraries, and systems biology often deal with massive datasets in a high dimensional space. The overarching goal of this dissertation is to enable fast and incremental similarity search over large high dimensional datasets through improved indexing, systematic heuristic optimizations, and scalable parallelization.

In Task 1, we design a sequential algorithm for All Pairs Similarity Search (APSS) that involves finding all pairs of records having similarity above a specified threshold. Our proposed fast matching technique speeds-up APSS computation by using novel tighter bounds for similarity computation and indexing data structure. It offers the fastest solution known to date with up to 6X speed-up over the state-of-the-art existing APSS algorithm.

In Task 2, we address the incremental formulation of the APSS problem, where APSS is performed multiple times over a given dataset while varying the similarity threshold. Our goal is to avoid redundant computations across multiple invocations of APSS by storing computation history during each APSS. Depending on the similarity threshold variation, our proposed history binning and index splitting techniques achieve speed-ups from 2X to over 10^5X over the state-of-the-art APSS algorithm. To the best of our knowledge, this is the first work that addresses this problem.

In Task 3, we design scalable parallel algorithms for APSS that take advantage of modern multi-processor, multi-core architectures to further scale-up the APSS computation. Our proposed index sharing technique divides the APSS computation into independent tasks and achieves ideal strong scaling behavior on shared memory architectures. We also propose a complementary incremental index sharing technique, which provides a memory-efficient parallel APSS solution while maintaining almost linear speed-up. Performance of our parallel APSS algorithms remains consistent for datasets of various sizes. To the best of our knowledge, this is the first work that explores parallelization for APSS.

We demonstrate the effectiveness of our techniques using four record datasets.
Fast, Incremental, and Scalable All Pairs Similarity Search

by
Amit Awekar

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2009

APPROVED BY:

Professor Kemafor Anyanwu
Professor Christopher G. Healey

Professor Nagiza F. Samatova
Chair of Advisory Committee
Professor Anatoli V. Melechko
Amit Awekar received a Bachelor of Engineering (BE) in Computer Engineering, with distinction, from Pune University, India in 2003. Amit was ranked in top one percentile computer science students in India. He completed his Master of Technology (MTech) degree in Computer Science and Engineering from Indian Institute of Technology, Kanpur, India in 2005. Amit was enrolled in North Carolina State University’s Ph.D. program in 8/2005. He successfully defended his Written Examination in 3/2007 and Oral Preliminary Examination in 8/2009. In the past, Amit has interned with Tata Institute of Fundamental Research, India and Yahoo! Research, India. He successfully completed the Certificate of Accomplishment in Teaching (CoAT) programme in 5/2008, and won the 2008 Outstanding Teaching Assistant Award from North Carolina State University.
ACKNOWLEDGMENTS

First, I would like to thank Professor Nagiza F. Samatova and Professor Anatoli V. Melechko, my advisors. They have been great mentors guiding me through my research. I am also indebted to Profs. Kemafor Anyanwu and Christopher G. Healey for serving on my thesis committee. I would like to thank Paul Breimyer and others in Prof. Samatova’s research group, for helping me time to time. I am also thankful to Bev Mufflemann for reviewing my dissertation draft. I want to thank my wife, Shraddha More, for all her love and trust. Finally, I am thankful to my parents, brother Ajit, and friends, for their continuous encouragement and support.

The work of Amit Awekar was funded by 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) through a contract with Oak Ridge National Laboratory (ORNL).
TABLE OF CONTENTS

LIST OF TABLES ......................................................... vi

LIST OF FIGURES ......................................................... vii

1 Introduction ......................................................... 1
   1.1 Motivations, Problems, and Contributions .................. 1
      1.1.1 Serial All Pairs Similarity Search (APSS) Algorithms . . . . 3
      1.1.2 Incremental APSS Algorithms ......................... 3
      1.1.3 Parallel APSS Algorithms .............................. 4
   1.2 Publications ................................................. 5

2 Serial APSS Algorithms ........................................... 6
   2.1 Introduction ................................................. 6
      2.1.1 Contributions ........................................ 7
      2.1.2 Results .............................................. 8
   2.2 Definitions and Notations ................................ 8
   2.3 Datasets .................................................... 9
   2.4 Common Framework ......................................... 12
      2.4.1 Preprocessing ........................................ 13
      2.4.2 Matching ............................................. 13
      2.4.3 Indexing .............................................. 14
   2.5 Fast Matching .............................................. 14
      2.5.1 Upper Bound on the Similarity Score ................... 15
      2.5.2 Lower Bound on the Candidate Vector Size ............ 15
   2.6 AP_Time_Efficient Algorithm ............................... 16
      2.6.1 Tighter Bounds ....................................... 16
      2.6.2 Effect of Fast Matching on Matching Phase ............ 17
   2.7 Extension to Tanimoto Coefficient ......................... 18
   2.8 Fast Matching Performance Evaluation ...................... 18
   2.9 Related Work ................................................ 23
   2.10 Conclusion ................................................. 24

3 Incremental APSS Algorithms ..................................... 28
   3.1 Introduction ................................................. 28
      3.1.1 Contributions ........................................ 30
      3.1.2 Results .............................................. 31
      3.1.3 IAPSS and Other Incremental Problems ................. 31
   3.2 Incremental APSS (IAPSS) Algorithm Overview .............. 31
   3.3 Booting .................................................... 32
      3.3.1 History Binning ...................................... 33
3.3.2 Booting Algorithm ........................................... 35
3.4 Upscaling ....................................................... 36
3.5 Downscaling .................................................... 40
  3.5.1 Division of Search Space ................................. 40
  3.5.2 Index Splitting ........................................... 40
  3.5.3 Downscaling Algorithm ................................. 41
3.6 Parallelization ................................................ 44
3.7 End-to-End IAPSS Performance ................................ 47
  3.7.1 Query Responsiveness to Similarity Value Change in IAPSS ..... 47
  3.7.2 Extreme Cases Speed-up .................................. 48
  3.7.3 Sensitivity to Varying $P_{max}$ .......................... 50
3.8 IAPSS with Reduced I/O Overhead ........................... 52
  3.8.1 Booting ..................................................... 53
  3.8.2 Upscaling and Downscaling .............................. 55
  3.8.3 Flooring ................................................... 55
  3.8.4 Extreme Cases Speed-up ................................. 56
3.9 Related Work .................................................. 56
3.10 Conclusion ..................................................... 57

4 Parallel APSS Algorithms ...................................... 58
  4.1 Introduction .................................................. 58
  4.1.1 Contributions ............................................. 61
  4.1.2 Results .................................................... 63
  4.2 Overview of Serial APSS Algorithms ........................ 63
  4.3 Index Sharing ................................................ 64
  4.3.1 Index Metadata Replication .............................. 65
  4.3.2 Static Partitioning ...................................... 65
  4.3.3 Dynamic Partitioning .................................... 68
  4.4 ISIMR Algorithm ............................................. 68
    4.4.1 ISIMR Algorithm Performance Evaluation ............... 69
  4.5 Incremental Index Sharing .................................. 78
  4.6 IISIC Algorithm ............................................. 78
    4.6.1 IISIC Algorithm Performance Evaluation ............... 79
  4.7 Related Work ................................................ 81
  4.8 Conclusion .................................................. 86

Bibliography ...................................................... 87
LIST OF TABLES

Table 2.1 Notations Used ......................................................... 10
Table 2.2 Datasets Used ............................................................ 10
LIST OF FIGURES

Figure 1.1 Thesis Overview .......................................................... 2
Figure 2.1 Unifying Framework for Recent Exact APSS Algorithms .......... 7
Figure 2.2 Distribution of Vector Sizes ............................................. 11
Figure 2.3 Distribution of Component Weights ................................ .... 12
Figure 2.4 Ignored Entries from Inverted Index vs. Similarity Threshold for Cosine Similarity ............................................................. 20
Figure 2.5 Candidate Pairs Evaluated vs Similarity Threshold for Cosine Similarity .. 21
Figure 2.6 Runtime in Seconds vs. Similarity Threshold for Cosine Similarity ..... 22
Figure 2.7 Speed-up Over All-Pairs Algorithm vs. Similarity Threshold for Cosine Similarity ............................................................. 23
Figure 3.1 IAPSS Overview .............................................................. 32
Figure 3.2 Running Time of IAPSS for the Booting Case ......................... 33
Figure 3.3 Size of Computation History for the Booting Case .................. 34
Figure 3.4 Running Time of IAPSS for the Upscaling Case ...................... 37
Figure 3.5 Speed-up of IAPSS for the Upscaling Case ............................ 38
Figure 3.6 Computation History Access Statistics for the Upscaling Case ...... 39
Figure 3.7 Parallelization Overview ................................................... 47
Figure 3.8 Running Time of IAPSS for the Downscaling Case ................. 48
Figure 3.9 Speed-up of IAPSS for the Downscaling Case ........................ 49
Figure 3.10 Fraction of Matching Pairs Immediately Found by Downscaling Algorithm 50
Figure 3.11 Best and Worst Case Speed-up for Similarity Values in Set T .......... 51
Figure 3.12 IAPSS Overview with Reduced I/O Overhead ........................ 53
Figure 3.13 Size of Computation History for Booting Case of IAPSS with Reduced I/O Overhead ............................................................ 54

Figure 4.1 Overview of Index Sharing Technique................................. 60
Figure 4.2 Overview of Incremental Index Sharing Technique ............... 62
Figure 4.3 Comparison of Various Partitioning Strategies .................... 67
Figure 4.4 Execution Time vs. Number of Processing Elements for ISIMR Algorithm and Serial APSS Algorithms ......................................................... 73
Figure 4.5 Speed-up Over All_Pairs Algorithm vs. Number of Processing Elements for ISIMR Algorithm ............................................................ 74
Figure 4.6 Speed-up Over AP_Time_Efficient Algorithm vs. Number of Processing Elements for ISIMR Algorithm ............................................................ 75
Figure 4.7 Efficiency vs. Number of Processing Elements for ISIMR Algorithm .... 76
Figure 4.8 Comparison of Speed-up of ISIMR Algorithm Over APT Algorithm for Different Dataset Sizes (n) in Multi-processor Environment ................. 77
Figure 4.9 Comparison of Efficiency of ISIMR Algorithm for Different Dataset Sizes (n) in Multi-processor Environment ......................................................... 77
Figure 4.10 Comparison of Memory Requirements of IISIC Algorithm with APT Algorithm ................................................................. 82
Figure 4.11 Comparison of Reduction in Memory Size for Various Datasets Using IISIC Algorithm ................................................................. 83
Figure 4.12 Execution Time vs. Number of Processing Elements for IISIC Algorithm on Multi-Processor Machine ......................................................... 84
Figure 4.13 Speed-up Over AP_Time_Efficient Algorithm vs. Number of Processing Elements for IISIC Algorithm on Multi-Processor Machine ................. 85
Chapter 1

Introduction

1.1 Motivations, Problems, and Contributions

With the exponentially growing size of available digital data and the limited human capacity to analyze data, search is one of the most important tools to cope with the resulting information overkill. Keyword search with a ranked output list of documents is the most popular search method from an end-user’s perspective [56]. To deliver this final result, a search system has to use various other types of search algorithms to pre-process [49, 29, 43], index [46, 45], mine [26, 76], and query [58] the data. For example, near duplicate search is used to clean the data by eliminating data instances that are almost duplicates [33, 47]; and k-nearest neighbor search is used to find anomalies in a dataset [78].

Many real-world systems frequently have to search for all pairs of records with similarity above the specified threshold. This problem is referred as the all pairs similarity search (APSS) [16], or similarity join [30]. Example applications that require one to solve the APSS problem include, but are not limited to:

- Web search engines that suggest similar web pages to improve user experiences [32];
- Online social networks that target similar users as potential candidates for new friendship and collaboration [74]; and
- Digital libraries that recommend similar publications for related readings [60].

Similarity between two data records (e.g., web pages, users, and publications) is defined via some similarity measure, such as the cosine similarity or the Tanimoto coefficient.
**APSS** is a compute-intensive problem. Given a dataset with \( n \) records in a \( d \)-dimensional space, where \( n << d \), the naïve algorithm for the APSS will compute the similarity between all pairs in \( O(n^2 \cdot d) \) time. Such a solution is not practical for datasets with millions of records, which are typical in real-world applications.

Thus, the research in this thesis is inspired by the need for the efficient treatment of the APSS problem for emerging and growing high-dimensional, sparse, and large datasets in Web-based applications like search engines, online social networks, and digital libraries. These applications use APSS to perform data mining tasks, such as collaborative filtering [48], near duplicate detection [72, 73, 65, 39, 40], coalition detection [62], clustering [20], and query refinement [70, 18]. Although these data mining tasks are not novel, the scale of the problem has increased dramatically, and the APSS problem is the rate limiting factor for their practical applicability.

![Figure 1.1: Thesis Overview](image-url)
In summary, solving the APSS problem for real-world applications is computationally challenging. This research focuses on designing and implementing scalable algorithms for solving the APSS problem within feasible time constraint. Please, refer to Figure 1.1 for thesis overview. Specifically, we propose the following advancements:

- Fast matching of data records in the APSS algorithms (Section 1.1.1 and Chapter 2);
- Incremental algorithm for the APSS problem with varying similarity thresholds (Sections 1.1.2 and Chapter 3); and
- Parallel APSS algorithm that further speeds-up APSS computation and enables processing of even larger datasets in a feasible time (Sections 1.1.3 and Chapter 4).

Next, we briefly summarize each of these advancements.

### 1.1.1 Serial All Pairs Similarity Search (APSS) Algorithms

We propose a unifying framework for existing APSS algorithms. The framework includes the following core components:

- data preprocessing, or sorting data records and computing summary statistics;
- pairs matching, or computing similarity between selective data record pairs; and
- record indexing, or adding a part of a data record to an indexing data structure.

Within this framework, we develop the fast matching technique by deriving novel tighter bounds on similarity computation and indexing structure. We incorporate these bounds into the state-of-the-art APSS algorithm to derive our AP.Time.Efficient algorithm, which is the fastest-to-date serial APSS algorithm.

Chapter 2 describes the fast matching technique and AP.Time.Efficient algorithm, and empirical studies demonstrating their superior performance. This work is published in the IEEE 2009 Web Intelligence Conference [12].

### 1.1.2 Incremental APSS Algorithms

Selecting a meaningful similarity threshold for APSS is an art because it is data dependent. Domain experts often use a trial-and-error approach by looking at the quality of output. 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.

Existing solutions for APSS fail to prune redundant computations across multiple invocations of APSS because each APSS is performed independently. We develop *history binning* and *index splitting* techniques for storing and clustering computation history during each invocation of APSS. The size of the computation history increases quadratically with the number of records in the dataset. This may create a significant I/O bottleneck. We introduce the concept of a *similarity floor* to store partial computation history, resulting in reduced I/O overhead. By incorporating these techniques into *AP.Time.Efficient* algorithm, we present the first solution for the IAPSS problem that systematically prunes redundant computations.

Chapter 3 discusses our IAPSS solution. This work is published as part of the ACM KDD, 2009 Workshop on Social Network Analysis and Mining [13]. Strategy to reduce I/O overhead in incremental APSS is published in the WorldComp 2009 International Conference on Information and Knowledge Engineering [14].

### 1.1.3 Parallel APSS Algorithms

Existing solutions for APSS are all limited to serial algorithms [71, 9, 16, 81, 79, 12]. This serial nature of APSS solutions is a limiting factor for applicability of APSS to large-scale real-world problems. Inspired by the success of parallel computing in dealing with large-scale problems [7, 67, 36, 15], we explore parallelization to further scale-up APSS computation.

To parallelize the APSS computation, we propose two complementary techniques: *index sharing* and *incremental index sharing*. Index sharing technique is designed with the goal of achieving linear speed-up over the fastest serial algorithm by dividing the APSS computation into independent tasks. However, for large datasets, the memory requirement of APSS might exceed the amount of main memory available. Incremental index sharing technique is designed to provide a memory-efficient parallel APSS solution while maintaining almost linear speed-up over the fastest serial algorithm.

Chapter 4 is devoted to parallel APSS algorithms. The manuscript summarizing this research is under peer review of a conference program committee.
1.2 Publications

1. **Title:** *Incremental All Pairs Similarity Search for Varying Similarity Thresholds with Reduced I/O Overhead* [14]
   Authors: **Amit Awekar**, Nagiza F. Samatova, and Paul Breimyer

2. **Title:** *Incremental All Pairs Similarity Search for Varying Similarity Thresholds* [13]
   The 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, the Third Workshop on Social Network Mining and Analysis, 2009. SNAKDD ’09, June 28, Paris, France.
   Authors: **Amit Awekar**, Nagiza F. Samatova, and Paul Breimyer

3. **Title:** *Fast Matching for All Pairs Similarity Search* [12]
   Authors: **Amit Awekar** and Nagiza F. Samatova

4. **Title:** *Selective Approach to Handling Topic Oriented Tasks on the World Wide Web* [10]
   The IEEE Symposium on Computational Intelligence and Data Mining, 2007. CIDM ’07, April 1-5, Honolulu, Hawaii, USA, (343-348).
   Authors: **Amit Awekar** and Jaewoo Kang

5. **Title:** *Selective Hypertext Induced Topic Search* [11]
   Authors: **Amit Awekar**, Jaewoo Kang, and Pabitra Mitra
Chapter 2

Serial APSS Algorithms

2.1 Introduction

In this chapter, we present \textit{AP\_Time\_Efficient} algorithm, which is the fastest serial algorithm for All Pairs Similarity Search (APSS). Our algorithm uses \textit{fast matching} technique that is based on novel tighter bounds for similarity computation and indexing data structure.

\textit{APSS} is a compute-intensive problem. Given a data set with \( n \) records in a \( d \)-dimensional space, where \( n << d \), the naive algorithm for the \textit{APSS} will compute the similarity between all pairs in \( O(n^2 \cdot d) \) time. Such a solution is not practical for data sets with millions of records, which are typical in real-world applications. Therefore, many heuristic solutions based on hashing [44, 28], shingling [23], or dimensionality reduction [51, 37] have been proposed to address this problem (please, refer to Section 2.9 for related work).

However, recent exact algorithms for the \textit{APSS} [71, 9, 16, 81, 79] have performed even faster than the heuristic methods because of their ability to significantly prune the similarity score computation by taking advantage of the fact that only a small fraction of all \( \Theta(n^2) \) pairs typically satisfy the specified similarity threshold. These exact algorithms depend on the \textit{inverted index}, which maps each dimension to the list of records with non-zero projection along that dimension.

We observe that the exact algorithms based on the inverted index share a common three-phase framework of:
• data preprocessing, or sorting data records and computing summary statistics;
• pairs matching, or computing similarity between selective data record pairs; and
• record indexing, or adding a part of data record to an indexing data structure.

![Unifying Framework for Recent Exact APSS Algorithms](image)

Figure 2.1: Unifying Framework for Recent Exact APSS Algorithms

Please, refer to Figure 2.1 for the framework overview. The preprocessing phase reorders the records and the components within each record based on some specified sort order, such as the maximum value or the number of non-zero components in the record. The matching phase identifies, for a given record, corresponding pairs with similarity above the threshold by querying the inverted index. The indexing phase then updates the inverted index with a part of the query record. The matching and indexing phases rely on filtering conditions and heuristic optimizations derived from the ordered records and components within each record.

The matching phase dominates the computational time. It searches for similar pairs in the inverted index and computes similarity score of pairs found. Therefore, improving the performance of any solution to the APSS would require optimization of these two tasks in the matching phase.

### 2.1.1 Contributions

Within the observed framework, we present the fast matching technique that reduces the effective size of the inverted index and the search space; the size of the search
space is defined as the actual number of record pairs evaluated by the algorithm. The proposed matching incorporates the following bounds:

- the lower bound on the number of non-zero components in any record, and
- the upper bound on the similarity score for any record pair.

The former allows for reducing the number of pairs that need to be evaluated, while the latter prunes, or only partially computes, the similarity score for many candidate pairs. Both bounds require only constant computation time. We integrate our fast matching technique with the fastest-to-date \textsl{All Pairs} algorithm [16] to derive the proposed \textsl{AP\_Time\_Efficient} algorithm.

2.1.2 Results

We conduct extensive empirical studies using four real-world million record data sets described in Section 2.3. We compare the performance of our \textsl{AP\_Time\_Efficient} algorithm against the state-of-the-art \textsl{All Pairs} algorithm [16] using two frequently used similarity measures: the cosine similarity and the Tanimoto coefficient. We achieve up to 6X speed up in our experiments, while reducing the search space by at least an order of magnitude.

2.2 Definitions and Notations

In this section, we define the \textsl{APSS} problem and other important terms referred throughout the thesis (please, see Table 2.1 for the summary of notations).

\textit{Definition 1 (All Pairs Similarity Search):} The all pairs similarity search (\textsl{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 (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 3 (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 4 (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:

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

Our algorithms can be extended to other popular similarity measures like the Tanimoto coefficient and the Jaccard similarity using transformations presented in Section 2.7.

### 2.3 Datasets

Out of the four datasets, one comes from the scientific literature collaboration information in Medline indexed papers [5]. The rest come from the popular online social networks, Flickr [2], LiveJournal [4], and Orkut [6]. These datasets were chosen, because they represent a variety of large-scale web-based applications like digital libraries and online social networks, that we are primarily interested in. The observed distribution of the vector sizes in the datasets is the power law distribution [64] (please, refer to Figure 2.2). These datasets are high dimensional and sparse (please, refer to Table 2.2). The ratio of the average size of vector 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 [16, 22]. These applications have to solve the APSS problem for high-dimensional datasets with millions of records; which are often sparse. Therefore, our algorithms will be relevant for other similar datasets as well.
Table 2.1: Notations Used

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{density}(j)</td>
<td>the number of vectors in ( V ) with non-zero projection along dimension ( j )</td>
</tr>
<tr>
<td>\textit{global_max_weight}[j]</td>
<td>( x[j] ) such that ( x[j] \geq y[j] ) for ( \forall y \in V )</td>
</tr>
</tbody>
</table>

Given a vector \( x \)

| \textit{x.max\_weight} | \( x[k] \) such that \( x[k] \geq x[i] \) for \( 1 \leq i \leq d \) |
| \textit{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 non-zero components in \( x \) |
| \( ||x|| \) (magnitude of \( x \)) | \( \sqrt{\sum_{i=1}^{d} x[i]^2} \) |

Given a pair of vectors \((x, y)\)

| \textit{dot}(x, y) | \( \sum_{i} x[i] \cdot y[i] \) |
| \textit{cos}(x, y) | \( \frac{\text{dot}(x, y)}{||x|| \cdot ||y||} \) |

Table 2.2: Datasets Used

<table>
<thead>
<tr>
<th>Dataset</th>
<th>( n = d )</th>
<th>Total Non-zero Components</th>
<th>Average Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medline</td>
<td>1565145</td>
<td>18722422</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>4598703</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>
Figure 2.2: Distribution of Vector Sizes

Medline

This dataset was selected to investigate possible applications for large web-based scientific digital libraries like PubMed, ACM Digital Library, and CiteSeer. Such digital libraries help users find similar publications and authors. We used the dataset prepared by the Auton Lab of Carnegie Mellon University. We are interested in finding pairs of authors that have similar collaboration patterns. Each vector represents the collaboration pattern of an author over the space of all authors. Two authors are considered to be collaborators if they write at least two papers together. Similar strategies were used in previous work [16] to eliminate accidental collaborations. We use the weighing scheme of Newman [66] to derive the weight of collaboration 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.
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 [74]. We use the dataset prepared by Mislove et al. [64]. Every user in the social network is represented by a vector over the space of all users. A user’s vector has non-zero projection along those dimensions that correspond to the users in his/her friend list. But the weights of these social network links are unknown. So, we applied the weight distribution from the Medline dataset to assign the weights to these social network links in the two datasets (please, refer to Figure 2.3). 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 and are available on the Web for downloading [1].

2.4 Common Framework

The basic idea behind the exact APSS algorithms based on the inverted index is similar to the way information retrieval systems answer queries [77, 75, 24]. Every vector in the data set is considered as a query and the corresponding matching pairs are found using
the inverted index. Most of the time, however, the information retrieval system requires only top-
\( k \) similar pairs [68, 69], while the APSS requires all matching pairs. The framework

\( \) can be broadly divided into three phases: data preprocessing, pairs matching, and indexing

(please, refer to Algorithm 1 for details).

2.4.1 Preprocessing

The preprocessing phase reorders vectors using a permutation \( \Omega \) defined over \( V \)

(lines 1-5, Algorithm 1). Bayardo et al. [16] and Xiao et al. [81] sorted vectors on the

maximum value within each vector. Sarawagi et al. [71] sorted vectors on their size. The

components within each vector are also rearranged using a permutation \( \Pi \) defined over \( D \).

Bayardo et al. [16] observed that sorting the dimensions in \( D \) based on vector density speeds

up the APSS. The summary statistics about each record, such as its size, magnitude, and

maximum component value are computed during the preprocessing phase. They are used

later to derive filtering conditions during the matching and indexing phases to save time

and memory. The time spent on preprocessing is negligible compared to the time spent on

matching.

2.4.2 Matching

The matching phase scans the lists in the inverted index that correspond to the

non-zero dimensions in \( x \), for a given vector \( x \in V \), to find candidate pairs (lines 7-16,

Algorithm 1). Simultaneously, it accumulates a partial similarity score for each candidate

pair. Bayardo et al. [16] and Xiao et al. [81] used the hash-based map, while Sarawagi et

al. [71] used the heap-based scheme for score accumulation.

Given \( t, \Omega, \Pi \) and summary statistics, various filtering conditions are derived to

eliminate candidate pairs that will definitely not satisfy the required similarity threshold;

these pairs are not added to the set \( C \) (line 7, Algorithm 1). Sarawagi et al. [71] identified

the part of the given vector \( x \in V \) such that for any candidate vector \( y \in V \) to have

\( \text{sim}(x,y) \geq t \), the intersection of \( y \) with that part must be non-empty. Bayardo et al. [16]

computed a lower bound on the size of any candidate vector to match with the current

vector as well as any remaining vector. Our fast matching technique further tightens this

lower bound.

Some of the candidate pairs can be safely discarded by computing an upper bound
on the similarity score in constant computational time. Xiao et al. [81] used the Hamming distance based method for computing such an upper bound. Bayardo et al. [16] used the vector size and maximum component value to derive a constant time upper bound. We further tighten this upper bound in our fast matching technique. Finally, the exact similarity score is computed for the remaining candidate pairs, and those having scores above the specified threshold are added to the output set.

2.4.3 Indexing

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 17-21, Algorithm 1). Sarawagi et al. [71] unconditionally indexed every component of each vector. Instead of building the inverted index incrementally, they built the complete inverted index beforehand. Bayardo et al. [16] and Xiao et al. [81] used the upper bound on the possible similarity score with only the part of the current vector. Once this bound reached the similarity threshold, the remaining vector components were indexed.

2.5 Fast Matching

In the framework described in Algorithm 1, the matching phase evaluates $O(n^2)$ candidate pairs. This phase is critical for the running time. The computation time of the matching phase can be reduced in three ways:

- traversing the inverted index faster while searching for candidate pairs,
- generating fewer candidate pairs for evaluation, and
- reducing the number of candidate pairs that are evaluated completely.

We propose two constant time optimizations that achieve these three goals. Our tighter lower bound on the size of the candidate vector reduces the effective size of the inverted index, and the number of candidate pairs that are being generated. Our tighter upper bound on the similarity score reduces the number of candidate pairs that are being evaluated completely. First, we will prove the correctness of these bounds, and later we will show that they are tighter than the existing bounds.
2.5.1 Upper Bound on the Similarity Score

Given a candidate pair \((x, y)\), the following constant time upper bound on the cosine similarity holds:

\[
\cos(x, y) \leq x_{\text{max weight}} \ast y_{\text{sum}}. \tag{2.1}
\]

The correctness of this upper bound can be derived from:

\[
x_{\text{max weight}} \ast \text{sum}(y) \geq \text{dot}(x, y) = \cos(x, y).
\]

Similarly following upper bound can be derived:

\[
\cos(x, y) \leq y_{\text{max weight}} \ast x_{\text{sum}}. \tag{2.2}
\]

Combining upper bounds in 2.1 and 2.2, we propose following upper bound on cosine similarity score:

\[
\text{min}(x_{\text{max weight}} \ast y_{\text{sum}}, y_{\text{max weight}} \ast x_{\text{sum}}), \tag{2.3}
\]

where \(\text{min}\) function selects minimum of the two arguments.
We can safely prune the exact dot product computation of a candidate pair if it does not satisfy the similarity threshold even for this upper bound.

2.5.2 Lower Bound on the Candidate Vector Size

For any candidate pair \((x, y)\), the following is true:

\[
x_{\text{max weight}} \ast y_{\text{sum}} \geq \text{dot}(x, y).
\]

For this candidate pair to qualify as a matching pair, the following inequality must hold:

\[
y_{\text{sum}} \geq t/x_{\text{max weight}}.
\]

For any unit-length vector \(y\), the following is true:

\[
y_{\text{sum}} \geq k \rightarrow |y| \geq k^2.
\]

This gives us the following lower bound on the size of any candidate vector \(y\):

\[
|y| \geq (t/x_{\text{max weight}})^2. \tag{2.4}
\]

The lower bound on the size of the candidate vector avoids generating the candidate pairs that will not satisfy the similarity threshold.
2.6 AP_Time_Efficient Algorithm

The proposed AP_Time_Efficient algorithm integrates both optimizations for the lower and upper bounds with the fastest-to-date All_Pairs algorithm [16] (please, refer to Algorithm 2). The vectors are sorted in decreasing order of their max_weight, and the dimensions are sorted in decreasing order of their vector density. For every $x \in V$, the algorithm first finds its matching pairs from the inverted index (Matching Phase, Lines 6-15) and then adds selective parts of $x$ to the inverted index (Indexing Phase, Lines 16-22). The difference between the algorithms AP_Time_Efficient and All_Pairs is in the tighter bounds on filtering conditions. The correctness of these tighter bounds is proven above. Hence, the correctness of the AP_Time_Efficient is the same as the correctness of the All_Pairs algorithm [16].

2.6.1 Tighter Bounds

The All_Pairs algorithm uses $t/x.max_weight$ as the lower bound on the size of any candidate vector. The AP_Time_Efficient algorithm tightens this bound by squaring the same ratio (Line 2, Algorithm 3).

The All_Pairs algorithm uses the following constant time upper bound on the dot product of any candidate pair $(x,y)$:

$$\min(|x|, |y|) \times x.max_weight \times y.max_weight.$$  \hfill (2.5)

We will consider following two possible cases to prove that our bound proposed in 2.3 is tighter than this bound.

1. Case 1: $|x| \leq |y|$

   The upper bound in 2.5 reduces to:

   $$|x| \times x.max_weight \times y.max_weight.$$

   The upper bound proposed in 2.2 is tighter than this bound because:

   $$x.sum \leq |x| \times x.max_weight$$

2. Case 2: $|x| > |y|$

   The upper bound in 2.5 reduces to:
\[ |y| \times x.text*weight \times y.text*weight. \]

The upper bound proposed in 2.1 is tighter than this bound because:

\[ y.text*sum \leq |y| \times y.text*max*weight \]

Hence, our upper bound proposed in 2.3 on the cosine similarity (Line 9, Algorithm 2) is tighter than the existing bound.

### 2.6.2 Effect of Fast Matching on Matching Phase

For a given vector \( x \), the lower bound on the size of a candidate vector is inversely proportional to \( x.text*max*weight \). The vectors are sorted in decreasing order by \( max*weight \). Hence, the value of the lower bound on the size of the candidate increases monotonically as the vectors are processed. If a vector \( y \) does not satisfy the lower bound on the size for the current vector, then it will not satisfy this bound for any of the remaining vectors. Such vectors can be discarded from the inverted index.

Like \( All:\text{Pairs} \), our implementation uses arrays for representing lists in the inverted index. Deleting an element from the beginning of a list will have linear time overhead. Instead of actually deleting such entries, we just ignore these entries by removing them from the front of the list (Line 4, Algorithm 3). Because of our tighter lower bound on the size of the candidate, more such entries from the inverted index are ignored. This reduces the effective size of the inverted index, thus, resulting in faster traversal of the inverted index during the finding of the candidate pairs. \( AP:\text{Time}Efficient \) also generates fewer candidate pairs as fewer vectors qualify to be candidate vectors because of the tighter lower bound on their size.

Computing the exact dot product for a candidate pair requires linear traversal of both vectors in the candidate pair (Line 10, Algorithm 2). The tighter constant time upper bound on the similarity score of a candidate pair prunes the exact dot product computation for a large number of candidate pairs. In our experiments we observed that \( AP:\text{Time}Efficient \) reduces the search space by up to two orders of magnitude (please, refer to Figure 2.5).

For the special case of binary vectors, where all non-zero values within a given vector are all equal, \( AP:\text{Time}Efficient \) will not provide any speed up over \( All:\text{Pairs} \).
because our optimizations depend on the variation in the values of different components within a given vector.

### 2.7 Extension to Tanimoto Coefficient

In this section, we extend \textit{AP\_Time\_Efficient} algorithm for the Tanimoto coefficient, which is similar to the extended Jaccard coefficient for binary vectors. Given a vector pair \((x, y)\), the Tanimoto coefficient is defined as:

\[
\tau(x, y) = \frac{\text{dot}(x, y)}{||x||^2 + ||y||^2 - \text{dot}(x, y)}.
\]

First, we will show that \(\tau(x, y) \leq \cos(x, y)\). Without loss of generality, let \(||x|| = a \cdot ||y||\) and \(a \geq 1\). By definition, it follows that:

\[
\frac{\cos(x, y)}{\tau(x, y)} = \frac{||x||^2 + ||y||^2 - \text{dot}(x, y)}{||x|| \ast ||y||},
\]

\[
= \frac{(a^2 + 1)/a - \cos(x, y)}{(a^2 + 1)/a - 1},
\]

\[
\geq 1 + (a - 1)^2/a,
\]

\[
\geq 1.
\]

Given the similarity threshold \(t\) for the Tanimoto coefficient, if we run both of our algorithms with the same threshold value for the cosine similarity, then they will not filter out any matching pairs. The only change we have to make is to replace the final similarity computation (Line 10 of Algorithms 2) with the actual Tanimoto coefficient computation as follows:

\[
s = \tau(x, y).
\]

### 2.8 Fast Matching Performance Evaluation

We performed experiments on four real-world data sets described in Section 2.3 for both the cosine similarity and the Tanimoto coefficient. Results for both similarity measures are quite similar. We present results only for cosine similarity for the sake of brevity. More details about results for the Tanimoto coefficient can be downloaded from the Web [1]. Many practical applications need pairs with relatively high similarity values [60, 74, 81, 32]. Hence, we varied the similarity threshold from 0.5 to 0.99 in 0.1 increments.
All our implementations are in C++. We used the standard template library for most of the data structures. We used the `dense_hash_map` class from Google\textsuperscript{TM} for the hash-based partial score accumulation [3]. The code was compiled using the GNU `gcc 4.2.4` compiler with \texttt{-O3} option for optimization. All the experiments were performed on the same 3 GHz Pentium-4 machine with 6 GB of main memory. The code and the data sets are available for download on the Web [1].

We evaluate the performance of fast matching in the `AP_Time_Efficient` algorithm based on three parameters:

- the size of the inverted index,
- the size of the search space, and
- the end-to-end run-time.

The preprocessing phase is identical in both algorithms. The time spent on data preprocessing was negligible as compared to the experiments’ running time, and was ignored. Though the indexing phase is also identical in both algorithms, the time spent in indexing phase was considered in the end-to-end running time, as it was not negligible.

The number of entries in the inverted index ignored by `AP_Time_Efficient` is two to eight times that of `All_Pairs` (please, refer to Figure 2.4). This reduction provides two-fold benefits: faster traversal of the inverted index and fewer candidate pairs evaluated. The memory footprints of `AP_Time_Efficient` and `All_Pairs` are the same because `AP_Time_Efficient` only ignores more entries from the inverted index, but does not remove them from memory. The `AP_Time_Efficient` reduces the search space by at least an order of magnitude (please, refer to Figure 2.5). Finally, the end-to-end speed up is up to 6X (please, refer to Figures 2.6 and 2.7). Even though the search space is reduced by an order of magnitude, we get comparatively less end-to-end speed up, because some time is still spent traversing the inverted index to find candidate pairs and filtering out a large fraction of them.

The best speed up is obtained for the Flickr data set, because it has a heavy tail in the distribution of vector sizes (please, refer to Figure 2.7). Vectors having the long size typically generate a large number of candidate pairs. But `AP_Time_Efficient` effectively avoids their generation and evaluation. In fact, for the Flickr data set, the number
Figure 2.4: Ignored Entries from Inverted Index vs. Similarity Threshold for Cosine Similarity
Figure 2.5: Candidate Pairs Evaluated vs Similarity Threshold for Cosine Similarity
Figure 2.6: Runtime in Seconds vs. Similarity Threshold for Cosine Similarity
Figure 2.7: Speed-up Over All_Pairs Algorithm vs. Similarity Threshold for Cosine Similarity

of candidate pairs evaluated by AP_Time_Efficient almost equals the actual number of matching pairs (please refer to Figure 2.5).

2.9 Related Work

Previous work on all pairs similarity search can be divided into two main categories: heuristic and exact. Main techniques employed by heuristic algorithms are hashing, shingling, and dimensionality reduction. Charikar [28] defines a hashing scheme as a distribution on a family of hash functions operating on a collection of vectors. For any two vectors, the probability that their hash values will be equal is proportional to their similarity. Fagin et al. [37] combined similarity scores from various voters, where each voter computes similarity using the projection of each vector on a random line. Broder et al. [23] use shingles and discard the most frequent features. Li et al. [59] use approximate string matching algorithms to evaluate similarity between tree structured data.

Exact algorithms are mainly inverted index based techniques. Sarawagi et al. [71] use efficient merging of lists in the inverted index, and clustering of related vectors to
reduce the computation time. Arasu et al. [9] generate signatures of input vectors and find pairs that have overlapping signatures. Finally, they output only those pairs that satisfy the similarity threshold. Xiao et al. [81] propose optimizations based on the length and Hamming distance specific to binary vectors. Xiao et al. [79] speed up similarity search based on edit distance measure for binary vectors. Bayardo et al. [16] propose various optimizations that employ the similarity threshold and sort order of the data, while finding matches and building the inverted index.

2.10 Conclusion

We described the inverted index based framework common across recent exact algorithms for all pairs similarity search. Within this framework, we presented tighter bounds on the candidate size and similarity score. These bounds reduced the search space by at least an order of magnitude and provided significant speed up for three large real-world data sets. Our fast matching technique will be likely relevant for other similar data sets, which are frequent across many web based systems.
Algorithm 1: Inverted Index Based Unifying Framework for Recent Exact APSS Algorithms

Input: $V$, $t$, $D$, $sim$, $\Omega$, $\Pi$

Output: \texttt{MATCHING_PAIRES_SET}

1. $\texttt{MATCHING_PAIRES_SET} = \emptyset$

2. $I_i = \emptyset$, $\forall 1 \leq i \leq d$
   
   /* The inverted index is initialized to $d$ empty lists. */

3. Arrange vectors in $V$ in the order defined by $\Omega$;

4. Arrange components in each vector in the order defined by $\Pi$;

5. Compute \texttt{summary_statistics};

6. foreach $x \in V$ using the order defined by $\Omega$ do

   7. $C = \text{set of candidate pairs corresponding to } x$, found by querying and manipulating the inverted index $I$;

   8. foreach candidate pair $(x, y) \in C$ do

      9. $\text{sim}_\text{max}_\text{possible} = \text{upper bound on } sim(x, y)$;

     10. if $\text{sim}_\text{max}_\text{possible} \geq t$ then

          11. $\text{sim}_\text{actual} = sim(x, y)$;

         12. if $\text{sim}_\text{actual} \geq t$ then

             13. $\texttt{MATCHING_PAIRES_SET} = \texttt{MATCHING_PAIRES_SET} \cup (x, y, \text{sim}_\text{actual})$

14.

15.

16. foreach $i$ such that $x[i] > 0$ using the order defined by $\Pi$ do

17. if $\text{filtering condition}(x[i])$ is true then

18. Add $(x, x[i])$ to the inverted index;

19.

20.

21. return $\texttt{MATCHING_PAIRES_SET}$
Algorithm 2: \textit{AP\_Time\_Efficient} Algorithm.

\textbf{Input}: $V, t, d, \text{global\_max\_weight}, \Omega, \Pi$

\textbf{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 $\text{max\_weight}$
4. $\Pi$ sorts dimensions in decreasing order by density
5. \textbf{foreach} $x \in V$ in the order defined by $\Omega$ \textbf{do}
   6. $\text{partScoreMap} = \emptyset$
      /* Empty map from vector id to partial similarity score */
   7. $\text{FindCandidates}(x, I, t, \Pi, \text{partScoreMap})$
   8. \textbf{foreach} $y: \text{partScoreMap}[y] > 0$ \textbf{do}
      9. \textbf{if} $\text{partScoreMap}[y] + \text{sum}(y) * x.\text{max\_weight} \geq t$ \textbf{then}
         /* Tighter upper bound on the similarity score */
         10. $s = \text{partScoreMap}[y] + \text{dot}(x, y')$;
         11. \textbf{if} $s \geq t$ \textbf{then}
            12. $\text{MATCHING\_PAIRS\_SET} = \text{MATCHING\_PAIRS\_SET} \cup (x, y, s)$
   13. \textbf{maxProduct} = 0;
   14. \textbf{foreach} $i: x[i] > 0$, in the order defined by $\Pi$ \textbf{do}
      15. $\text{maxProduct} = \\
         \text{maxProduct} + x[i] * \text{min(\text{global\_max\_weight}[i], x.\text{max\_weight}})$;
      16. \textbf{if} $\text{maxProduct} \geq t$ \textbf{then}
         17. $I_i = I_i \cup \{x, x[i]\}$;
         18. $x[i] = 0$;
   19. \textbf{return} $MPS$
Algorithm 3: FindCandidates Algorithm.

Input: $x, I, t, \Pi, \text{partScoreMap}$

Output: modified $\text{partScoreMap}$, and $I$

\[
\text{remMaxScore} = \sum_{i=1}^{d} x[i] \times \text{global\_max\_weight}[i];
\]

\[\text{minSize} = \left(\frac{t}{x.\text{max\_weight}}\right)^2;\]

\[/* \text{Tighter lower bound on candidate size} */

foreach $i$: $x[i] > 0$, in the reverse order defined by $\Pi$ do

Iteratively remove $(y, y[i])$ from front of $I_i$ while $|y| < \text{minSize}$;

foreach $(y, y[i]) \in I_i$ do

if $\text{partScoreMap}(y) > 0$ or $\text{remMaxScore} \geq t$ then

\[
\text{partScoreMap}(y) = \text{partScoreMap}(y) + x[i] \times y[i];
\]

\[
\text{remMaxScore} = \text{remMaxScore} - \text{global\_maximum\_weight}[i] \times x[i];
\]

/* Remaining maximum score that can be added after processing current dimension */
Chapter 3

Incremental APSS Algorithms

3.1 Introduction

In Chapter 2, we presented APSS algorithms for a fixed similarity threshold value. However, selecting a meaningful similarity threshold for all pairs similarity search (APSS) 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 Jarvis-Patrick clustering algorithm sparsifies the similarity score matrix by retaining only those entries that satisfy a predefined threshold [52]. The optimal threshold for sparsifying the similarity score matrix 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) that 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. For example, to detect near duplicate documents [81], 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.

To the best of our knowledge, the IAPSS problem has not received a special treatment in scientific literature and the “brute-force” strategy is used instead [80]. 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 [81, 16, 9, 71] 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.

For a dataset with $n$ records in a $d$ dimensional space where $d >> n$, a brute-force algorithm for IAPSS will compute the similarity scores between all possible $O(n^2)$ pairs during each of the $s$ selections for the similarity threshold value. As each similarity computation requires $O(d)$ time, the total time complexity will be $O(n^2 \times s \times d)$. 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:

- developing efficient I/O techniques to deal with possibly large history data; and
- 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 recomputation 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 \( t_{\text{new}} \) and the previous lowest similarity threshold value \( t_{\text{old}} \), 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 \( t_{\text{old}} \leq t_{\text{new}} \), and

3. **downscaling**, where \( t_{\text{old}} > t_{\text{new}} \).

The history binning technique is used in all three cases, while index splitting is required only for the downscaling case.

### 3.1.1 Contributions

We incorporate both history binning and index splitting into the *Time-Efficient* algorithm, which is the state-of-the-art APSS algorithm [12]. This incorporation enables us to split the IAPSS computation into various independent subtasks that can be executed in parallel. This chapter 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.
3.1.2 Results

We perform empirical studies using four real-world million record datasets described in Section 2.3. All experiments were performed on a 2.6 GHz Intel\textsuperscript{TM} Xeon\textsuperscript{TM} class machine with eight CPU cores and 16 GB of main memory. We compare the performance of our algorithm against the \textit{All_Pairs} algorithm [16] and \textit{AP_Time_Efficient} algorithm [12]. Depending on the similarity threshold variation, our speed-ups vary from $2X$ to over $10^5X$.

3.1.3 \textit{IAPSS} and Other Incremental Problems

The \textit{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 [82]. Unlike these incremental methods, the \textit{IAPSS} problem assumes that such datasets remained unchanged across different searches. Some incremental algorithms are designed to identify the \textit{top-k} similar pairs [27, 83, 80]. But the \textit{IAPSS} problem requires all \textit{matching pairs}. Incremental algorithms for the distance join [50, 61] address problems similar to \textit{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.

3.2 Incremental \textit{APSS} (\textit{IAPSS}) Algorithm Overview

The \textit{IAPSS} problem is to solve the \textit{APSS} problem for a given similarity threshold value $t_{\text{new}}$ when the \textit{APSS} problem has already been solved for the previous value of similarity threshold $t_{\text{old}}$. The \textit{IAPSS} algorithm is based on the observation that the proportion of the search space explored during the execution of a single \textit{APSS} invocation is inversely proportional to the value of the similarity threshold. If $t < t'$, then the search space explored while executing \textit{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 \textit{IAPSS} problem. Depending on the relative values of the current similarity threshold ($t_{\text{new}}$) and the previous lowest similarity threshold value ($t_{\text{old}}$), Figure 3.1 gives
an overview of the IAPSS algorithm and there are three 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.

### 3.3 Booting

Booting is a relatively simple case of IAPSS that performs APSS while recording the computation history using history binning.
3.3.1 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. For example, if $P_{\text{max}} = 5$, 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 $P_s = \lfloor s \cdot P_{\text{max}} \rfloor$. For the special case of $s = 1$ the partition number is $P_{\text{max}} - 1$. All experiments reported in this paper are performed with $P_{\text{max}} = 20$. The effect of varying $P_{\text{max}}$ is discussed in Section 3.7.3.

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 2). 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 2). These pairs are stored in an exact pairs file of the partition corresponding to their exact similarity score.

Figure 3.2: Running Time of IAPSS for the Booting Case
3.3.2 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 Candidate Pair Producer executes the All_Pairs algorithm (please, refer to Algorithm 4), and the Candidate Pair Consumer writes candidate pairs to persistent storage (please, refer to Algorithm 5).

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 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 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.
Algorithm 4: Candidate Pair Producer Algorithm: Replace lines 9-13 of Algorithm 2 with the Following Pseudocode

```
upperBound =
    partScoreMap{y} + min(sum(y')*x.max_weight, sum(x)*y'.max_weight);
if upperBound ≥ t then
    s = partScoreMap{y} + dot(x,y');
    Add (x,y,s,true) to candidatePairQueue;
    if s ≥ t then
        MPS = MPS ∪ (x,y,s)
    else
        Add (x,y,upperBound,false) to candidatePairQueue;
```

Figure 3.2 shows the running time of the IAPSS booting algorithm for various similarity threshold values. Performance of IAPSS for the booting case is the same as the AP.Time.Efficient algorithm.

### 3.4 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 4). 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 \( t_{new} \). 
Algorithm 5: Candidate Pair Consumer Algorithm for Writing Candidate Pairs to Persistent Storage

1. while doneFlag not true do
   2. Dequeue all candidate pairs from candidatePairsQueue in writePairsSet;
   3. foreach Element w in writePairsSet do
      4. \[ P_w = \lfloor w.score \times P_{\text{max}} \rfloor; \]
      5. if w.isExact is true then
         6. Append entry \((w.x, w.y, w.score)\) to file for exact pairs corresponding to partition \(P_w\)
      7. else
         8. Append entry \((w.x, w.y)\) to file for approximate pairs corresponding to partition \(P_w\)

and then reads the exact pairs files corresponding to all partitions \(P, P_{\text{new}} \leq P < P_{\text{max}}\). The pairs satisfying the threshold \(t_{\text{new}}\) 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 3.4); this is expected because the algorithm only reads and outputs matching pairs. It results in large speed-ups in the range \(10^2 X\) to \(10^6 X\) (please, refer to Figure 3.5). The speed-up for the upscaling case is not dependent on the value \(t_{\text{old}}\) because the number of pairs read by the upscaling algorithm depends only on the value of \(t_{\text{new}}\).

Grouping pairs by similarity score enables our algorithm to only read the required portions of the history. Figure 3.6 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.
3.5 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.

3.5.1 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 $t_{new}$.
(a) Speed-up Over *AllPairs*

(b) Speed-up Over *AP_Time_Time_Efficient*

Figure 3.5: Speed-up of *IAPSS* for the Upscaling Case
Figure 3.6: Computation History Access Statistics for the Upscaling Case
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 $C_{known} = C - C_{unknown} = C_{low} \cup C_{match}$.

3.5.2 Index Splitting

The size of the inverted index is inversely proportional to the value of the similarity threshold (lines 16-21, Algorithm 2). 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 $C_{unknown}$.

3.5.3 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.

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 $P_{new} \leq P < P_{max}$ (Algorithm 7). This step is similar to the upscaling case.
Algorithm 6: Downscaling Algorithm.

**Input:** \( V, t, d, \text{global\_max\_weight}, \Omega, \Pi, P_{max} \)

**Output:** \( MPS \) (Matching Pairs Set)

1. \( MPS = \emptyset, I_i^{old} = \emptyset, \forall 1 \leq i \leq d \), \( I_i^{new} = \emptyset, \forall 1 \leq i \leq d \);

2. \hspace{1em} \textbf{foreach} \( x \in V \) in the order defined by \( \Omega \) \textbf{do}
   3. \hspace{2em} \text{Initialize} \( \text{approxList} \) and \( \text{knownList} \) to empty sets;
   4. \hspace{2em} \textbf{ReadCMatch}();
   5. \hspace{2em} \textbf{foreach} \( \text{Partition} \ P : P_{new} \leq P < P_{max} \) \textbf{do}
      6. \hspace{3em} \text{Add each} \( y \) to \( \text{ApproxList} \), such that \((x,y)\) is approximate pair in \( P \);
      7. \hspace{3em} \text{Delete} \( (x,y) \) from computation history;
   8. \hspace{2em} \text{FindKnownCandidates}();
   9. \hspace{2em} \text{FindNewCandidates}(x,I,t);  
   10. \hspace{2em} \textbf{foreach} \( y : \text{partScoreMap}\{y\} > 0 \) \textbf{do}
       11. \hspace{3em} \text{upperBound} = \text{partScoreMap}\{y\} + \min(\text{sum}(y') * \text{x\_max\_weight}, \text{sum}(x) * y'\_max\_weight);
       12. \hspace{3em} \textbf{if} \text{upperBound} \geq t \textbf{then}
           13. \hspace{4em} \text{s} = \text{partScoreMap}\{y\} + \text{dot}(x,y');
           14. \hspace{4em} \text{Add} \((x,y,s,\text{true})\) to \text{candidatePairQueue};
           15. \hspace{4em} \textbf{if} \text{s} \geq t \textbf{then}
               16. \hspace{5em} \text{MPS} = \text{MPS} \cup (x,y,s)
           17. \hspace{4em} \text{else}
               18. \hspace{5em} \text{Add} \((x,y,\text{upperBound},\text{false})\) to \text{candidatePairQueue};
       19. \hspace{3em} \text{SplitIndexVector}();
   20. \hspace{2em} \text{t}_{old} = \text{t}_{new};
   21. \hspace{2em} \text{store updated value of} \text{t}_{old} \text{to persistent storage};
   22. \hspace{1em} \text{return} \text{MPS}
Algorithm 7: ReadCMatch Algorithm for Reading Pairs in $C_{\text{match}}$

1. foreach Partition $P : P_{\text{new}} \leq P < P_{\text{max}}$ do
2.     foreach Exact Pair $(x,y)$ in partition $P$ do
3.         /* $s = \text{sim}(x,y)$. Read from computation history. */
4.         if $s \geq t_{\text{new}}$ then
5.             MPS = MPS \cup (x,y,s);

Reading and Evaluating $C_{\text{approx}}$

Similar to the pairs in $C_{\text{match}}$, pairs in $C_{\text{approx}}$ 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 $C_{\text{approx}}$ during the matching phase (lines 15-16, Algorithm 6). For a given vector $x$, the list of pairs in $C_{\text{approx}}$ is stored in $\text{approxList}$. The partial similarity score for these pairs is calculated using the inverted index when finding $C_{\text{known}}$ and $C_{\text{new}}$ (please, refer to procedures $\text{FindKnownCandidates}$ and $\text{FindNewCandidates}$). The similarity score computation using the inverted index is more efficient than serially traversing the vectors. In addition, the evaluation for $C_{\text{approx}}$ now piggybacks searching of $C_{\text{known}}$ and $C_{\text{new}}$.

Finding $C_{\text{known}}$

Finding all the pairs in $C_{\text{known}}$ can be accomplished by reading the entire computation history. However, finding $C_{\text{known}}$ from the inverted index is more efficient because it is an in-memory data structure. For a given vector $x$, the $\text{FindKnownCandidates}$ procedure finds pairs in $C_{\text{known}}$. It traverses the inverted index in the same manner as the $\text{FindCandidates}$ procedure in Algorithm 2. However, the similarity score is computed only for pairs in the $\text{approxList}$. The list of pairs in $C_{\text{known}}$ is stored in the $\text{knownList}$. 
Algorithm 8: SplitIndexVector Algorithm

Input: $x$, $I^{old}$, $I^{new}$, $t^{old}$, $t^{new}$, $\Pi$

Output:

1. $maxProduct = 0$;
2. foreach $i$: $x[i] > 0$, in the order defined by $\Pi$ do
   3. $maxProduct = maxProduct + x[i] \cdot \min(\text{global}_{max\_weight}[i], x_{max\_weight})$;
   4. if $maxProduct \geq t^{old}$ then
      5. $I^{old}_i = I^{old}_i \cup \{x, x[i]\}$;
      6. $x[i] = 0$;
   7. else
      8. if $maxProduct \geq t^{new}$ then
         9. $I^{new}_i = I^{new}_i \cup \{x, x[i]\}$;
         10. $x[i] = 0$;

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 2. 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 $C_{unknown}$.

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 13-23, Algorithm 6).
Algorithm 9: FindKnownCandidates Algorithm

Input: $x$, $I^{old}$, $t^{old}$, $\Pi$, partScoreMap, knownList, approxList

Output: modified partScoreMap, and knownList

1. $\text{partScoreMap} = \emptyset$;
2. $\text{remMaxScore} = \sum_{i=1}^{d} x[i] \ast \text{global\_max\_weight}[i]$;
3. $\text{minSize}_{old} = \left(\frac{t^{old}}{x.\text{max\_weight}}\right)^2$;
4. foreach $i: x[i] \gt 0$, in the reverse order defined by $\Pi$ do
   5. Iteratively ignore $(y, y[i])$ from front of $I^{old}_i$ while $|y| < \text{minSize}_{old}$;
   6. foreach $(y, y[i]) \in I^{old}_i$ do
      7. if $y \in \text{approxList}$ then
         8. $\text{partScoreMap}[y] = \text{partScoreMap}[y] + x[i] \ast y[i]$;
      else
         9. Add $y$ to knownList;
      10. $\text{remMaxScore} = \text{remMaxScore} - \text{global\_maximum\_weight}[i] \ast x[i]$;
   11. if $\text{remMaxScore} < t^{old}$ then
      12. return

3.6 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 6. However, various smaller subtasks presented in Section 3.5.3 present opportunities for parallelizing the downscaling computation. These subtasks can run in parallel, while data flows through these subtasks.

Figure 3.7 shows the parallelization outline. It works as a pipeline of producers
Algorithm 10: FindNewCandidates Algorithm

Input: $x$, $I^{old}$, $t_{old}$, $\Pi$, partScoreMap, knownList, approxList

Output: modified partScoreMap

\[ remMaxScore = \sum_{i=1}^{d} x[i] \times global\_max\_weight[i]; \]

\[ minSize_{old} = (t_{old}/x.max\_weight)^2; \]

\[ minSize_{new} = (t_{new}/x.max\_weight)^2; \]

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 $|y| < minSize_{old}$ do

if $y \notin knownList$ then

partScoreMap{$y$} = partScoreMap{$y$} + $x[i] \times y[i];$

endforeach $(y, y[i]) \in I_i^{new}$ do

if $y \notin knownList$ then

partScoreMap{$y$} = partScoreMap{$y$} + $x[i] \times y[i];$

endforeach $(y, y[i]) \in I_i^{new} \cup I_i^{old}$ do

if $y \notin knownList$ then

if partScoreMap{$y$} > 0 or remMaxScore $\geq t_{new}$ then

partScoreMap{$y$} = partScoreMap{$y$} + $x[i] \times y[i];$

remMaxScore = remMaxScore - global\_maximum\_weight[i] \times x[i];
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 4 and 5. For other producer-consumer pairs, synchronization scheme is similar.

Figure 3.8 running time of the IAPSS downscaling case. We started with a booting similarity threshold of 0.99. Then we reduced the similarity threshold to 0.5 in 0.1 decrement steps. Speed-up over All_Pairs and Ap_Time_Efficient is plotted in Figure 3.9.
3.7 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:

\[ T = \{0.99, 0.9, 0.8, 0.7, 0.6, 0.5\} \]

3.7.1 Query Responsiveness to Similarity Value Change 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

---

Figure 3.8: Running Time of IAPSS for the Downscaling Case
Figure 3.9: Speed-up of IAPSS for the Downscaling Case
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 3.10 shows the ratio of the number of pairs in $C_{match}$ to the total number of matching pairs for various downscaling similarity threshold values. This ratio represents the part of the output immediately generated by the downscaling algorithm.

![Figure 3.10: Fraction of Matching Pairs Immediately Found by Downscaling Algorithm](image)

### 3.7.2 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.

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

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 3.11 shows the best and worst case speed-ups achieved by the IAPSS solution compared to the All_Pairs algorithm and AP_Time_Efficient 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.

### 3.7.3 Sensitivity to Varying $P_{\text{max}}$

The $P_{\text{max}}$ parameter is used to divide the similarity range into equal sized partitions. For a given value of $t_{\text{new}}$, the IAPSS algorithm has to read the computation history for all partitions $P : P_{\text{new}} \leq P < P_{\text{max}}$. Some pairs in the partition $P_{\text{new}}$ will not satisfy the similarity threshold, but must be read anyway. This overhead is attenuated if the partition floor equals $t_{\text{new}}$, i.e., $P_{\text{new}} = t_{\text{new}} \times P_{\text{max}}$. However, we observed that this overhead is not significant. During our experiments, we varied the $P_{\text{max}}$ 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.

### 3.8 IAPSS with Reduced I/O Overhead

The size of the computation history increases quadratically with the number of records in the dataset. We introduce the concept of a similarity floor ($\text{simFloor}$) to store only partial computation history, resulting in reduced I/O overhead. 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 IAPSS invocation then their similarity score is recomputed. We observed that the size of the computation history is reduced by at least an order of magnitude for our datasets.
Figure 3.11: Best and Worst Case Speed-up for Similarity Values in Set $T$
Figure 3.12: IAPSS Overview with Reduced I/O Overhead

Figure 3.12 shows the overview of IAPSS solution with reduced I/O overhead. Depending on the relative values of the current similarity threshold ($t_{new}$), the previous lowest similarity threshold value ($t_{old}$), and similarity floor ($simFloor$), there are four possible cases for the IAPSS solution:

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.

4. **Flooring**: $t_{new} < simFloor$, deleting all computation history and performing IAPSS like the booting case.
3.8.1 Booting

Booting case remains the same except for the introduction of \( \text{simFloor} \) parameter, 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 11). 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 3.13). The running time for the booting case remains the same for all datasets. For the booting case, the value of \( \text{simFloor} \) has to be less than or equal to \( t_{new} \). All pairs with similarity score at least \( t_{new} \) must be present in the computation history for upscaling and downscaling.

Algorithm 11: Candidate Pair Producer with Reduced I/O Overhead:
Following Pseudocode Replaces Lines 9-13 of Algorithm 2

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


Figure 3.13: Size of Computation History for Booting Case of IAPSS with Reduced I/O Overhead
3.8.2 Upscaling and Downscaling

Upscaling and downscaling cases are not affected by introduction of simFloor. Reduction in the size of computation history is transparent for these two cases. Any pair with similarity score less than simFloor is not required for these two cases.

Upscaling case reads a pair only if it has similarity score at least \( t_{\text{new}} \). All pairs required for upscaling case are already in computation history because \( \text{simFloor} \leq t_{\text{old}} \leq t_{\text{new}} \). For downscaling case, eliminated part of the computation history is a subset of \( C_{\text{low}} \). Downscaling algorithm ignores all pairs in \( C_{\text{low}} \).

3.8.3 Flooring

The flooring case is similar to the downscaling case with exception that \( \text{simFloor} > t_{\text{new}} \). 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 \( \text{simFloor} > t_{\text{new}} \), some pairs in \( C_{\text{match}} \) and \( C_{\text{approx}} \) will have similarity score less than \( \text{simFloor} \). 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.

3.8.4 Extreme Cases Speed-up

If IAPSS algorithm with reduced I/O overhead, 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.

For the best case, IAPSS algorithm with reduced I/O, will perform like simple IAPSS algorithm. For the worst case, performance will be like the \( AP\_Time\_Efficient \) algorithm.
3.9 Related Work

To the best of our knowledge, there is no direct previous work on the IAPSS problem. Many other problems, such as Top-$K$ similarity join (TKS) and K-Nearest Neighbors (KNN) are similar to the IAPSS problem, and a rich body of work on incremental algorithms exists for such problems. However, techniques used by those algorithms cannot be applied to the IAPSS problem because of the differences elaborated in Section 3.1.3.

TKS is the problem of finding top $k$ ranked matching pairs in a given dataset. Xiao et al. [80] borrow prefix filtering principle from APSS algorithms [16, 81]. Their approach is based on new pruning and optimization techniques for finding top-$k$ matching by leveraging monotonocity of maximum possible similarity score of unseen pairs and $k$th largest similarity value seen so far. Top-$K$ spatial join problem involves retrieval of the $top-k$ objects satisfying a spatial constraint [83]. However, algorithms for this problem deal with datasets with dimensionality and distance functions satisfying the triangle inequality. Incremental algorithms for the distance join [50, 61] have the same limitations.

KNN is the problem of finding top-$k$ nearest data instances for a given data instance. Existing solutions mainly use space partitioning based data structures like KD Trees [34] and R Trees [17] are used to arrange points in a given dataset. KNN search is then run in two steps: (1) finding leaf node that will contain the query point, and (2) backtracking the tree upwards to find $k$ nearest neighbors. Recent empirical comparison of various space partitioning methods [53] shows that no particular data structure offers significant speed up over other for various data sets. However, experience from past research of two decades on KNN search suggests, that tree based methods are a good choice for small dimensionality only [19].

3.10 Conclusion

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.

• Selective storage of the computation history reduced I/O overhead by at least an order of magnitude.

The compounded effect of these approaches resulted in speed-ups of $2X$ to over $10^5X$ on four large-scale real-world datasets.
Chapter 4

Parallel APSS Algorithms

4.1 Introduction

In this chapter, we present parallel algorithms for APSS. We explore the two proposed complementary strategies for APSS parallelization: (1) index sharing and (2) incremental index sharing.

Remind that All Pairs Similarity Search (APSS) is the problem of searching for all pairs of data records with the similarity between two records above the specified threshold. Given a dataset with \( n \) data records in a \( d \) dimensional space, where \( n \ll d \), existing APSS algorithms compute \( O(n^2) \) similarity scores, while searching through \( O(n \times d) \) size inverted index that maps each dimension to a list of data records that have a non-zero projection along that dimension. Thus, the nature of the existing APSS solutions is compute-, as well as, data-intensive.

Existing solutions for APSS are all limited to serial algorithms [71, 9, 16, 81, 79, 12]. The compute- and data-intensive nature of APSS is a rate limiting factor for applicability of APSS to large-scale real-world problems and calls for alternative approaches. Processor clock rates are not expected to increase dramatically in the near future [42]. With the emergence of shared memory multi-processor, multi-core architectures, parallel algorithms that take advantage of such emerging architectures are a promising strategy. Throughout this chapter, we will use the term processor to refer to a single processor or a processing core within a multi-core processor, unless stated otherwise. Inspired by the success of parallel computing in dealing with large-scale problems [7, 67, 36, 15], we explore parallelization to further speed-up APSS computation.
Parallel algorithms for APSS should enable processing of large datasets in a reasonable amount of time. Web-based applications like search engines, online social networks, and digital libraries are increasingly dealing with more massive datasets [35, 55]. Without scalable parallel APSS algorithms, it will likely not be practical to run APSS over some of these applications’ datasets, which are growing at an exponential rate [63, 31].

It is generally believed that a scalable, parallel solution for APSS will effectively help design scalable, parallel solutions for important data mining tasks like clustering and collaborative filtering that use APSS as their underlying operator. Middlewares like pR [21] can use parallel solution for APSS to speed-up statistical analysis algorithms.

We expect that a parallel solution for APSS will help further speed-up our solution for incremental APSS (IAPSS) problem presented in Chapter 3. Shared memory parallel algorithms presented for IAPSS have been designed for the incremental formulation of the APSS problem with varying similarity thresholds. However, when APSS gets executed for the first time during the booting phase, the IAPSS performance is the same as the performance of the serial APSS algorithm. Our IAPSS solution becomes attractive only if APSS is executed frequently over the same dataset. However, serial execution of the first APSS invocation is still a bottleneck for the IAPSS solution. This bottleneck can be overcome by using a parallel APSS algorithm during the booting phase.

The compute- and data-intensive nature of the APSS problem poses the following technical challenges for its parallel solution:

- The inverted index must be shared across all processors and updated incrementally.
- The huge size of the dataset and of the inverted index makes data transfers between processors prohibitively expensive.

We address these challenges by proposing two complementary techniques: index sharing and incremental index sharing. Please, refer to Figures 4.1 and 4.2 for the overview of these techniques. The index sharing technique is designed with the goal of achieving linear speed-up over the fastest serial algorithm at the cost of a slightly larger memory footprint. However, for large datasets, the memory requirement of APSS might exceed the amount of main memory available. Therefore, the incremental index sharing technique is designed to provide a memory-efficient parallel APSS solution, while maintaining an almost linear speed-up over the fastest serial algorithm.
(a) Indexing Phase

(b) Matching Phase

Figure 4.1: Overview of Index Sharing Technique
The index sharing technique parallelizes the APSS matching phase, which is described in Section 2.4, into independent searches over the central inverted index, which is shared as a read-only data structure across all processors. Index sharing builds the central inverted index before starting the matching phase, unlike existing APSS algorithms that incrementally build the inverted index during the matching phase. Each processor keeps and updates its own copy of index metadata of reasonably small size, thus, resulting in a slightly larger memory footprint. A subset of data records is assigned to each processor to perform the matching phase. We explore various static and dynamic strategies for distributing data records among processors.

Incremental index sharing reduces the memory footprint of APSS solution by incrementally building the inverted index and maintaining only a subset of the inverted index in the processor main memory. Incremental index sharing divides the inverted index into non-overlapping subsets of roughly equal size using the Round-Robin partitioning strategy described in Section 4.5. Only one subset of the inverted index is incrementally constructed and maintained in memory. The index sharing technique is used to perform the matching phase for all data records over the target subset of the inverted index, which is shared across all processors as a read-only data structure. Once this subset gets processed, it is discarded from memory, and the same procedure is repeated for the next target subset of the inverted index.

4.1.1 Contributions

To the best of our knowledge, this is the first work that explores parallelization for APSS. We propose the following contributions:

- We develop the index sharing and the incremental index sharing techniques to parallelize the AP.Time.Efficient algorithm, which is the fastest serial algorithm for APSS.

- Our index sharing based parallel APSS algorithm achieves linear speed-up in a shared memory environment.

- Our incremental index sharing based parallel APSS algorithm achieves up to 32% memory savings while maintaining almost speed-up in a shared memory environment.

- We provide scalable solutions to perform APSS over large datasets in reasonable time.
Figure 4.2: Overview of Incremental Index Sharing Technique

(a) Indexing Phase

(b) Matching Phase
4.1.2 Results

We perform empirical studies using four real-world, million record datasets described in Section 2.3. We compare the performance of our algorithms against \textit{All.Pairs} algorithm [16] and the \textit{AP.Time.Efficient} algorithm [12]. All our experiments are performed in a shared memory environment.

Our index sharing based \textit{ISIMR} algorithm achieves super-linear speed-up over the \textit{All.Pairs} algorithm and linear speed-up over the \textit{AP.Time.Efficient} algorithm. The performance of the \textit{ISIMR} algorithm remains consistent for datasets of various sizes. Our incremental index sharing based \textit{IISIC} algorithm achieves up-to 32% memory savings over both \textit{All.Pairs} and \textit{AP.Time.Efficient} algorithms, while maintaining almost linear speed-up.

4.2 Overview of Serial APSS Algorithms

All existing algorithms for APSS are serial in nature [71, 9, 16, 81, 79, 12]. Recent exact algorithms for APSS depend on the inverted index. Remind that these exact algorithms share a common three-phase framework of:

- data preprocessing, or sorting data records and computing summary statistics;
- pairs matching, or computing similarity between selective data record pairs; and
- record indexing, or adding a part of data record to an indexing data structure.

The preprocessing phase reorders data records and record components using various attributes such as: the number of components in a data record or the maximum component value within a data record. Then, the matching phase identifies, for a given record, corresponding pairs with the similarity above the specified threshold by querying the inverted index. The matching phase also removes redundant entries from the inverted index (Chapter 2, Algorithm 2, Lines 6-15). The indexing phase then adds a part of the given data record to the inverted index (Chapter 2, Algorithm 2, Lines 16-22). The matching phase dominates the computing time of APSS, and the time spent during preprocessing and indexing is negligible. Please, refer to Section 2.4 for a detailed description of the common framework.
The \textit{AP\_Time\_Efficient} algorithm is also based on the common framework described above and is the fastest serial algorithm for \textit{APSS}. Please, refer to Section 2.6 for more detailed description of \textit{AP\_Time\_Efficient} algorithm.

4.3 Index Sharing

The index sharing technique is based on parallelizing the matching phase of \textit{APSS} into independent searches over the central inverted index which is shared across all processors as a read-only data structure. In contrast, the data records are partitioned among processors to perform the matching phase. We explore both static and dynamic partitioning strategies. Each processor performs the matching phase independently of other processors using the central inverted index.

Read and write access to the inverted index during the matching phase is the major bottleneck in parallelizing the \textit{AP\_Time\_Efficient} algorithm. This bottleneck arises because of the following two reasons:

1. \textit{AP\_Time\_Efficient} algorithm adds a given data record to the inverted index only after performing the matching phase for that record (Chapter 2, Algorithm 2, Lines 16-22). Thus, new entries are added to the inverted index during the matching phase.

2. The matching phase in the \textit{AP\_Time\_Efficient} algorithm updates the inverted index during each search by discarding redundant entries from the inverted index (Algorithm 3, Line 4).

If the matching phase in a parallel \textit{APSS} algorithm requires read and write access to the central inverted index, then each processor will require exclusive access to the inverted index, resulting in \textit{synchronization overheads}. Index sharing overcomes this bottleneck by building the inverted index before starting the matching phase and by replicating the reasonably small size index metadata across all processors.

To perform the matching phase for a given data record, all the data records with the ids prior to the given record’s id must be present in the inverted index. Hence, the \textit{AP\_Time\_Efficient} algorithm cannot be adopted directly to perform search for multiple data records simultaneously. Index sharing overcomes this limitation by building the whole inverted index before starting the matching phase.
4.3.1 Index Metadata Replication

Index metadata is the set of start offset values maintained, one for each list of data record ids in the inverted index, indicating the front of the list. Index sharing replicates the index metadata across all processors to eliminate the need for any synchronization among processors while performing the matching phase. For a dataset with $n$ data records in a $d$ dimensional space, the inverted index contains $O(n \times d)$ entries, while the size of the metadata is only $O(d)$. Compared to the size of the inverted index, the size of the metadata is reasonably small and grows linearly with the number of dimensions.

The matching phase of the $\text{AP\_Time\_Efficient}$ algorithm requires write access to the inverted index to discard redundant entries from the inverted index. The preprocessing phase in the $\text{AP\_Time\_Efficient}$ algorithm sorts data records in the decreasing order of the maximum value of any component within the record. Using this sort order, the $\text{AP\_Time\_Efficient}$ algorithm derives a lower bound on the size of data records in the inverted index to match with any of the remaining data records. While performing the matching phase for a given record, the entries that correspond to data records not satisfying the lower bound on their size are discarded from the inverted index.

For time efficiency purposes, the $\text{AP\_Time\_Efficient}$ algorithm does not actually remove the redundant entries from the inverted index, but only ignores them using the index metadata. The $\text{AP\_Time\_Efficient}$ algorithm uses arrays for representing lists in the inverted index. Deleting an element from the beginning of a list will have linear time overhead. Instead of actually deleting such entries, the algorithm simply ignores these entries by removing them from the front of the list (Chapter 2, Algorithm 3, Line 4). The start offset corresponding to an inverted list array is incrementally advanced as entries are removed from the front.

Index sharing replicates the index metadata across all processors to eliminate the need for synchronization between processors while performing the matching phase. Each processor updates its local index metadata after performing the matching phase for every data record assigned to it.

4.3.2 Static Partitioning

The goal of the index sharing technique is to divide the computation workload of the matching phase roughly equally across all processors. We consider two static partition-
ing strategies: block partitioning and Round-Robin partitioning.

**Block Partitioning**

The block partitioning strategy assigns a contiguous block of data records to each processor. The time required to perform the matching phase for a given data record increases as *APSS* proceeds from the beginning of the dataset to the end. This variation arises because the preprocessing phase puts short data records, i.e. records with fewer number of non-zero components at the beginning of the dataset. Compared to short data records, longer data records require more time to perform the matching phase because they generate comparatively more candidate pairs for evaluation.

Due to the variation in the time required for the matching phase of data records, assigning equal number of contiguous data records to each processor will likely create severe work imbalance among the processors. The block partitioning strategy tries to compensate this imbalance by assigning an equal number of components to each processor. If short data records are assigned to a processor, then that processor will have more number of data records assigned than a processor with longer data records assigned.

**Round-Robin Partitioning**

The block partitioning strategy assigns all short data records to initial partitions and all longer data records to later partitions, resulting in a severe imbalance in the distribution of the computation workload of the matching phase across various processors. Please, refer to Figure 4.3 for an example of this imbalance. This example was generated while running index sharing based parallel *APSS* algorithm using block partitioning strategy for the Orkut dataset.

Work imbalance induced by the block partitioning strategy can be reduced by assigning data records to each processor in a Round-Robin fashion. If there are *P* processors, then any consecutive *P* data records in the dataset are assigned to a different processor by the Round-Robin partitioning strategy. Please, refer to Figure 4.3 for an example of the performance improvement achieved in evenly distributing the computation workload of the matching phase by the Round-Robin strategy over the block partitioning strategy.
Load Balancing across Multiple Processors

(a) Different Processors’ Execution Times for Various Partitioning Strategies

Mean and Standard Deviation of Processor Execution Times

(b) Mean and Standard Deviation of Processors’ Execution Times for Various Partitioning Strategies

End-to-end Running Time

(c) End-to-end Running Time for Various Partitioning Strategies

Figure 4.3: Comparison of Various Partitioning Strategies
4.3.3 Dynamic Partitioning

The nature of the APSS problem makes it impossible to determine a priori a well-balanced distribution of data records among processors. The amount of time required for the matching phase of different data records, even of the same size, can vary depending on the effectiveness of filtering conditions described in Section 2.5 and the number of candidate pairs to evaluate. Both of these factors are impossible to predict. Thus, certain processors might take longer than others to process assigned data records. Without dynamic load balancing, the execution time of different processors can differ greatly. An example of this imbalance is shown in Figure 4.3.

Dynamic partitioning strategy aims at maximizing the processor utilization efficiency by dynamically assigning a small batch of data records to a processor as soon as the corresponding processor finishes the previous batch. As a result, all processors are expected to finish their computation almost the same time. Please, refer to Figure 4.3 for an example of the performance improvement achieved by evenly distributing the computation workload of the matching phase via the dynamic partitioning strategy over static partitioning strategies.

4.4 ISIMR Algorithm

The Index Sharing with Index Metadata Replication (ISIMR) algorithm parallelizes the AP_Time_Efficient algorithm by incorporating the index sharing technique (Algorithm 12). This is a shared memory parallel algorithm. Please, refer to Table 2.1 for the summary of notations used.

The ISIMR algorithm can be divided into three phases: data preprocessing, record indexing, and pairs matching. The preprocessing phase is the same as in the AP_Time_Efficient algorithm (Algorithm 12, Lines 3 and 4). Before starting the matching phase, the indexing phase builds the entire inverted index, which is globally shared across all processors as a read-only data structure (Algorithm 13).

The matching phase of the ISIMR algorithm is executed in parallel. Please, refer to Algorithm 14. The ISIMR algorithm uses the dynamic partitioning strategy, which results in a balanced distribution of the computation workload of the matching phase across all processors. Each time a processor finishes its current work, a batch_size number of data
records are assigned to the corresponding processor to perform the matching phase. The work assignment is not carried out by any explicit master. All processors keep track of the number of data records processed through a shared variable, called \textit{last\_processed\_record}. Every time a new batch of data records is assigned to a processor, this variable is incremented by \textit{batch\_size}. The matching phase terminates when all data records are processed.

\subsection*{4.4.1 ISIMR Algorithm Performance Evaluation}

We performed experiments on four real-world datasets described in Section 2.3 for both the cosine similarity and the Tanimoto coefficient measures. Results for both similarity measures are quite similar. We present results only for cosine similarity for the sake of brevity. More details about the results for the Tanimoto coefficient can be downloaded from the Web [1]. The results presented here are an aggregate of experiments performed by varying the similarity threshold value from 1.0 to 0.5 in decrements of 0.1. The time spent for preprocessing and indexing is negligible as compared to the time spent for the matching phase. In all our experiments, we consider the time required only for the matching phase.

All of our implementations are for shared memory environment and coded in C++. We implement parallelization using the POSIX Pthreads library [57]. We used the standard template library for most of the data structures. We used the dense hash map class from Google\textsuperscript{TM} for the hash-based partial score accumulation [3]. The code was compiled using the GNU gcc 4.2.4 compiler with -O3 option for optimization. The experiments were performed on multi-processor as well as multi-core shared memory computers, each with eight processing elements. The code and the datasets are available for download on the Web [1].

As described in Section 4.1, motivation for parallelizing APSS is to create a solution that scales with the number of processing elements as well as with the size of datasets. Therefore, we evaluate the performance of the ISIMR algorithm based on scalability with respect to the number of processing elements and to the number of data records. The \textit{ISIMR} algorithm achieves ideal performance for both metrics.

In Figure 4.4, the running time of the \textit{ISIMR} algorithm is compared to the running time of \textit{All\_Pairs} and \textit{AP\_Time\_Efficient} algorithms. For multi-processor environment, \textit{ISIMR} achieves ideal strong scaling behavior, i.e. linear speed-up over the \textit{AP\_Time\_Efficient} algorithm (please, refer to Figures 4.6 and 4.7).
Algorithm 12: ISIMR Algorithm.

**Input:** $V$, $t$, $d$, global_max_weight[], $\Omega$, $\Pi$, $P$, batch_size

- $V$ = Set of data vectors
- $t$ = Similarity threshold
- $d$ = Set of dimensions
- global_max_weight[] = Array containing maximum value in each dimension across all data records
- $\Omega$ = Function that sorts vectors in the decreasing order of max_weight
- $\Pi$ = Function that sorts dimensions in the decreasing order of their density
- $P$ = Number of processing elements
- batch_size = Number of elements assigned at-a-time to a processing element

**Output:** MATCHING_PAIRS_SET

1. MATCHING_PAIRS_SET = $\emptyset$;
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. BuildIndex();
6. last_processed_record=0;
7. for $i=0$; $i<P$; $i++$ do
   8. Spawn thread $T_i$;
   9. Have $T_i$ run ProcessPartition();
10. Wait for all $P$ threads to finish;
11. return MATCHING_PAIRS_SET
Algorithm 13: BuildIndex Algorithm.

1. foreach $x \in V$ in the order defined by $\Omega$ do
2.     $\text{maxProduct} = 0$;
3.     foreach $i$: $x[i] > 0$, in the order defined by $\Pi$ do
4.         $\text{maxProduct} =$
5.             $\text{maxProduct} + x[i] \times \min(\text{global max weight}[i], x.\text{max weight})$;
6.         if $\text{maxProduct} \geq t$ then
7.             $I_i = I_i \cup \{x, x[i]\}$;

The performance of the ISIMR algorithm degrades in multi-core environment due to cache thrashing and memory bandwidth limitation [25]. In our experiments, the size of the datasets and of the inverted index range from few hundred megabytes to multiple gigabytes. ISIMR and other APSS algorithms access the inverted index and the dataset randomly, thus, resulting in cache thrashing. Multi-core environment has multiple processing cores, but they still share bus connection to the shared memory. When more cores start competing for memory access, the ISIMR performance degrades. Thrashing effect is more visible for larger datasets like Orkut, while linear speed-up is maintained for smaller datasets, like Medline.

Our ISIMR algorithm achieves superlinear speed-up over the All_Pairs algorithm (please, refer to Figure 4.5). This superlinear scaling is the combined effect of tighter filtering conditions used by the ISIMR algorithm during the matching phase (please, refer to Section 2.5) and the index sharing technique.

Scalability of ISIMR with respect to variations in dataset sizes is plotted in Figures 4.8 and 4.9. The performance of the ISIMR algorithm remains consistent. This result suggests that ISIMR algorithm will likely scale well with other large datasets.
Algorithm 14: ProcessPartition Algorithm.

1 Initialize each element of startOffset[d] array to 0;
   /* Local index metadata */
2 while do
   mutex_lock;
   if last_processed_record >= total_records then
   mutex_unlock;
   return;
   local_startRecord = last_processed_record;
   local_endRecord = last_processed_record;
   mutex_unlock;
   foreach x ∈ V from local_startRecord to local_endRecord do
      partScoreMap = ∅;
      /* Empty map from vector id to partial similarity score */
      FindCandidates(x, I, t, Π, partScoreMap, startOffset);
      /* Chapter 2, Algorithm 3 */
   foreach y: partScoreMap[y] > 0 do
      if partScoreMap[y] + sum(y) * x.max_weight ≥ t then
         s = partScoreMap[y] + dot(x, y');
         if s ≥ t then
            MATCHING_PAIRS_SET =
            MATCHING_PAIRS_SET ∪ (x, y, s)
Figure 4.4: Execution Time vs. Number of Processing Elements for ISIMR Algorithm and Serial APSS Algorithms
Figure 4.5: Speed-up Over *All_Pairs* Algorithm vs. Number of Processing Elements for *ISIMR* Algorithm
Figure 4.6: Speed-up Over *AP.Time_Efficient* Algorithm vs. Number of Processing Elements for *ISIMR* Algorithm
Figure 4.7: Efficiency vs. Number of Processing Elements for ISIMR Algorithm
Figure 4.8: Comparison of Speed-up of ISIMR Algorithm Over APT Algorithm for Different Dataset Sizes ($n$) in Multi-processor Environment

Figure 4.9: Comparison of Efficiency of ISIMR Algorithm for Different Dataset Sizes ($n$) in Multi-processor Environment
4.5 Incremental Index Sharing

The incremental Index Sharing technique is designed with the goal of performing parallel APSS with reduced memory footprint while maintaining speed-up over serial algorithms. Existing solutions for APSS, including the ISIMR algorithm described in the previous section, assume that the whole inverted index and the dataset can fit in the main memory. This assumption might not be valid for some systems, especially while processing large datasets. In such cases, incremental index sharing technique provides an effective alternative memory-efficient parallel solution.

The incremental index sharing technique divides the inverted index into non-overlapping partitions. The inverted index is built incrementally and only one partition of the inverted index is maintained in the main memory. Index sharing technique is used to perform the matching phase for all data records over the current index partition in the memory. Then the current index partition is discarded and other index partitions are processed in the same manner. Please refer to Figure 4.2 for the overview of incremental index sharing.

The number of partitions is an input parameter to the incremental index sharing technique. For a given number of partitions, the smallest memory footprint is achieved by creating all index partitions of the same size. To create equal sized index partitions, incremental index sharing assigns data records to each index partition in a Round-Robin fashion. If there are a total $P$ index partitions, then any consecutive $P$ data records in the dataset are each assigned to a different index partition by the Round-Robin partitioning strategy.

4.6 IISIC Algorithm

The Incremental Index Sharing with Incremental Construction (IISIC) algorithm parallelizes the AP_Time_Efficient algorithm by incorporating the incremental index sharing technique (Algorithm 15). This is a shared memory algorithm. Please, refer to Table 2.1 for the summary of notations used.

The IISIC algorithm can be divided into three phases: data preprocessing, record indexing, and pairs matching. The preprocessing phase is the same as in the APT algorithm (Algorithm 15, Lines 2 and 3). Before starting the matching phase, the indexing phase builds
a subset of the inverted index using the Round-Robin partitioning strategy for selecting data records (Algorithm 16). This subset of the inverted index is shared across all processors as a read-only data structure (Algorithm 15, Lines 8-11). The matching phase of the IISIC algorithm is executed in parallel like the ISIMR algorithm. Other index partitions are processed in the same manner.

4.6.1 IISIC Algorithm Performance Evaluation

Datasets and experimental set-up for the performance evaluation of the IISIC algorithm are the same as those of the ISIMR algorithm described in Section 4.4.1.

As described in Section 4.5, the motivation for developing the IISIC algorithm is to provide a memory-efficient parallel APSS solution while maintaining scalable speed-up over serial algorithms. Therefore, we evaluate performance of the ISIMR algorithm based on scalability with respect to the number of processing elements and to the number of index partitions.

In Figure 4.10, the IISIC algorithm is compared with the AP_Time_Efficient algorithm based on the memory footprint size. Memory footprint size decreases as the number of index partitions are increased. The IISIC algorithm achieves up-to 32% memory savings for eight index partitions (please, refer to Figure 4.11).

The AP_Time_Efficient algorithm keeps both the dataset and the inverted index in memory. The IISIC algorithm reduces the memory footprint by keeping only a part of the inverted index in memory, while the whole dataset is still in memory. The IISIC algorithm achieves better memory savings, if the ratio of the size of the inverted index to the size of the dataset increases. This ratio increases if data records are longer, i.e., there are more non-zero components per data record. Therefore, the performance of the IISIC in terms of memory savings is the best for the Orkut dataset (with the average data record size of 75), while the performance degrades for the Medline dataset (with the average data record size of 11).

In Figure 4.12, the running time of the IISIC algorithm is compared to the running time of the AP_Time_Efficient algorithm for multi-processor environment. The IISIC running time increases as the number of index partitions are increased. This increase in the running time is due to the overhead of performing the matching phase for all data records multiple times using each index partition. However, the IISIC algorithm still
Algorithm 15: IISIC Algorithm.

Input: $V$, $t$, $d$, $\text{global\_max\_weight}[\ ]$, $\Omega$, $\Pi$, $P$, $\text{batch\_size}$, $\text{num\_partitions}$

/* $V$ = Set of data vectors */
/* $t$ = Similarity threshold */
/* $d$ = Set of dimensions */
/* $\text{global\_max\_weight}[\ ]$ = Array containing maximum value in each dimension across all data records */
/* $\Omega$ = Function that sorts vectors in the decreasing order of $\text{max\_weight}$ */
/* $\Pi$ = Function that sorts dimension in the decreasing order of their density */
/* $P$ = Number of processing elements */
/* $\text{batch\_size}$ = Number of elements assigned at-a-time to a processing element */
/* $\text{num\_partitions}$ = Number of partitions of the inverted index */

Output: $\text{MATCHING\_PAIRS\_SET}$

MATCHING\_PAIRS\_SET = $\emptyset$;

$\Omega$ sorts vectors in decreasing order by $\text{max\_weight}$;

$\Pi$ sorts dimensions in decreasing order by density;

for $i=0$; $i<\text{num\_partitions}$; $i++$ do

$I_i = \emptyset$, $\forall 1 \leq i \leq d$;

/* Initialize the inverted index to all empty lists. */

BuildIndexPartition ($i$);

last\_processed\_record=0;

for $i=0$; $i<P$; $i++$ do

Spawn thread $T_i$;

Have $T_i$ run ProcessPartition ($\_\_\_\_$);

Wait for all $P$ threads to finish;

return MATCHING\_PAIRS\_SET
Algorithm 16: BuildIndexPartition Algorithm.

Input: partitionId

1 for x=partitionId; x < |V|; x = x + num_partitions do
    /* Assign data records to index partitions in a Round-Robin fashion. */
    maxProduct = 0;
    foreach i: x[i] > 0, in the order defined by Π do
        maxProduct =
        maxProduct + x[i] * min(global_max_weight[i], x.max_weight);
        if maxProduct ≥ t then
            I_i = I_i ∪ {x, x[i]};

maintains an almost linear speed-up over the AP.Time.Efficient algorithm (please, refer to Figure 4.13).

4.7 Related Work

To the best of our knowledge, there is no direct previous work on parallel algorithms for the APSS problem. Research in the past decade has primarily focused on improving APSS solutions through novel indexing schemes, heuristic optimizations, dimensionality reduction, and tighter bounds on similarity computation [12, 81, 16, 9, 71, 37, 28, 44, 51]. However, parallelization has been explored for other related problems, such as Top-K similarity join (TKS) and K-Nearest Neighbors (KNN).

All distance calculations in KNN search are independent of each other and can be executed in parallel. Garcia and Debreuve [41] propose parallelization of this naïve algorithm using Graphics Processing Units (GPUs). They parallelize the space partitioning based KNN search methods and propose various heuristic optimizations for speed-up. For even larger datasets and peer-to-peer (P2P) environments, distributed algorithms are necessary. Falchi et al. [38] propose KNN search scheme for distributed environments. For
Figure 4.10: Comparison of Memory Requirements of IISIC Algorithm with APT Algorithm
Figure 4.11: Comparison of Reduction in Memory Size for Various Datasets Using *IISIC* Algorithm
Figure 4.12: Execution Time vs. Number of Processing Elements for IISIC Algorithm on Multi-Processor Machine
Figure 4.13: Speed-up Over \textit{AP\_Time\_Efficient} Algorithm vs. Number of Processing Elements for \textit{IISIC} Algorithm on Multi-Processor Machine
a given query, each node performs local $KNN$ search and then communicates the summary of local results to other nodes.

For distance based similarity joins, Alsabti et al. [8] propose parallel algorithms using $e-k-d$-$BTree$ data structure, which is a space partitioning tree. Each processor builds a local $e-k-d$-$BTree$ using the assigned data partition. Each processor then identifies overlapping regions with other processors, and exchanges the required data points, followed by local similarity joins. Lieberman et al. [61] transform the distance based similarity join problem as a sort and search problem, which can be solved efficiently and in parallel on GPUs. They introduce a data structure called $Z$-list that maps all data points to natural numbers. In a $d$ dimensional space, $d + 1$ such lists are created in parallel and kept sorted. For each query data point, intervals in each list that might contain a point in the given distance range are identified in parallel.

Kim and Lee [54] propose parallel algorithms for the record linkage problem that involves finding and recursively merging similar records. Their approach is based on data partitioning and concurrent merge operations across data partitions. However, they do not address the problem of efficient similarity computation and focus only on merging strategies.

### 4.8 Conclusion

We presented scalable, parallel solutions for the APSS problem based on two proposed techniques: index sharing techniques and incremental index sharing. The ISIMR algorithm, which is an index sharing based parallel APSS algorithm, achieves ideal strong scaling performance and this performance remains consistent with variations in the dataset sizes. The IISIC algorithm, which is an incremental index sharing based parallel APSS algorithm achieves up to 32% memory savings over other APSS algorithms while maintaining almost linear speed-up over the state of the art serial APSS algorithm. The performance of both algorithms presented in this chapter suggests that APSS can be performed over large datasets in reasonable time using parallelization.
Bibliography


References


