Optimizing Parallel Algorithms for All Pairs Similarity Search

Maha Alabduljalil, Xun Tang, Tao Yang
University of California at Santa Barbara, CA 93106, USA
{maha,xtang,tyang}@cs.ucsb.edu

ABSTRACT

All pairs similarity search is used in many web search and data mining applications. Previous work has used comparison filtering, inverted indexing, and parallel accumulation of partial intermediate results to expedite its execution. However, shuffling intermediate results can incur significant communication overhead as data scales up. This paper studies a scalable two-step approach called Partition-based Similarity Search (PSS) which incorporates several optimization techniques. First, PSS uses a static partitioning algorithm that places dissimilar vectors into different groups and balance the comparison workload with a circular assignment. Second, PSS executes comparison tasks in parallel, each using a hybrid data structure that combines the advantages of forward and inverted indexing. Our evaluation results show that the proposed approach leads to an early elimination of unnecessary I/O and data communication while sustaining parallel efficiency. As a result, it improves performance by an order of magnitude when dealing with large datasets.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Search Process, Clustering; H.3.4 [**Systems and Software**]: Distributed Systems

General Terms

Parallel Algorithms, Performance

Keywords

Similarity search, partitioning, load balancing, filtering

1. INTRODUCTION

All Pairs Similarity Search (APSS) is a core function for many data-intensive applications that identifies a set of objects similar to another under a specified threshold. Examples of such applications include: collaborative filtering for similarity-based recommendations [1], search query suggestions [19], coalition detection for advertisement frauds [17], spam detection [9, 15, 14], web mirrors and plagiarism [21], and near duplicate detection [13, 15].

Conducting similarity is a time-consuming process, especially when a massive amount of data is involved. Previous research on expediting the process has used filtering methods and inverted indexing to eliminate unnecessary computations[8, 23, 2]. However, parallelization of such methods is not straightforward given the extensive amount of I/O and communication overhead involved. One popular approach is the use of MapReduce [10] to compute and collect similarity results in parallel using an inverted index [16, 18]. In addition, computation filtering can be applied to avoid unnecessary computation [7]. Unfortunately, the cost of communicating the intermediate partial results is still excessive and such solutions are un-scalable for larger datasets.

Thus we pursue a design that conducts partition-based APSS in parallel without the need for reducer tasks. Furthermore, to avoid unnecessary data loading and comparison, we statically group data vectors into partitions such that the dissimilarity of partitions is revealed in an early stage. In addition, previous works [8, 23, 2, 18, 7] have used characteristics of the partially computed similarity scores to filter out similarity comparisons. These methods can be viewed as dynamic filtering, since they happen at runtime. On the other hand, our approach is a fast static partitioning method that divides the dataset in a way that dissimilar documents are more likely mapped to different machines.

The proposed partitioning performs an early removal of unwanted comparisons which eliminates a significant amount of unnecessary I/O, memory access and computation. We further present a circular partition-based comparison assignment that exploits the computation symmetry and balances the parallel workload. In designing data structure and computation flow of each parallel task, we investigate the tradeoffs between different choices for data access. We found that a hybrid combination of forward and inverted indexing can lead to a significantly better performance. The proposed two-step method, called Partition-based Similarity Search (PSS), is more scalable than previously proposed methods and can lead to an order-of-magnitude improvement in performance when dealing with large datasets.

The rest of this paper is organized as follows. Section 2 reviews background and the problem statement. Section 3 gives an overview of PSS. Section 4 describes the partitioning algorithm and the circular load balancing scheme. Section 5 presents the execution of PSS with a hybrid index data structure. Section 6 presents the experimental results and Section 7 concludes this paper.

WSDM'13, February 4–8, 2013, Rome, Italy.

Task Map(feature t, posting $P(t)$) :
For each pair of vectors $(d_i, d_j) \in P(t)$ **do**
Emit $(d_i, d_j, w_{i,t} \times w_{j,t})$.

Task $Reduce(d_i, d_j$, partial scores $s_1, s_2, ...)$:
Sum partial scores $s_1 + s_2 + ..$ as $Sim(d_i, d_j)$;
Write $(d_i, d_j, Sim(d_i, d_j))$ if $Sim(d_i, d_j) \geq \tau$.

Figure 1: MapReduce tasks with inverted index for APSS.

2. BACKGROUND

Following the definition in [8], assume a set of n real-valued vectors $D = \{d_1, d_2, \cdots, d_n\}$, and each vector contains at most m non-negative features, $d_i = \{w_{i,1}, w_{i,2}, \cdots, w_{i,m}\}$. For simplicity, each vector is normalized and hence the cosine-based similarity $Sim(d_i, d_j)$ is computed as:

$$Sim(d_i, d_j) = \cos(d_i, d_j) = \sum_{t \in (d_i \cap d_j)} w_{i,t} \times w_{j,t}.$$

Two vectors x, y are considered similar if their similarity score exceeds a threshold τ, namely $Sim(x, y) \geq \tau$.

In many problem domains, for example web text data, input vectors are sparse and a vast majority of feature weights are zero. We denote the 1-norm value of a vector as $\|d_i\|_1$, which is the summation of all its non-zero values. Denote $maxw[d_i]$ to be the maximum feature weight in vector d_i, which is also known as $\|d_i\|_\infty$.

Given n vectors with k features on average per vector, a naïve algorithm computes similarity scores with complexity $O(n^2k)$. While threshold-based computation filtering [20, 6] lowers computation cost, APSS is still expensive for large datasets such as web documents.

Inverted and forward indexing have been studied in [16] to compute top-k similar vectors using MapReduce. Figure 1 shows the pseudo-code of the map and reduce tasks for a simplified APSS algorithm with feature-centric inverted index. Each feature-centric map is assigned a unique set of postings and derives partial similarity scores for all pairs of documents sharing a feature. Reducers then merge these partial scores for each vector pair. Parallel APSS has been studied in [5] for shared memory multi-processor machines where the inverted index is shared among processors to reduce inter-processor communications. There are further techniques to optimize the inverted index code performance: 1) only compute pairs of vectors (d_i, d_j) such that $i < j$ due to the symmetry nature; 2) scale down the size of the map task outputs so that each reduce task handles the partial scores between a document d_i and all other documents [16]; 3) eliminate participation of features that have only one document ID in their postings; and 4) filter additional computations as discussed in the next paragraph. However, even with extensive optimization to filter unnecessary computation, map-reduce communication with the inverted index can increase quadratically as the dataset size scales up and can incur significant synchronization and I/O overhead.

Computation-filtering techniques have also been developed to continuously monitor partially accumulated similarity scores and dynamically detect dissimilar pair of documents. This bases on the given similarity threshold without complete derivation of the similarity value [8, 23]. The key formula used for this filtering is:

$$Sim(d_i, d_j) = \sum_{t \in \chi} w_{i,t} \times w_{j,t} + \sum_{t \in (d_i \cap d_j - \chi)} w_{i,t} \times w_{j,t}. \quad (1)$$

Here, set χ contains the common features that have been scanned and their contributed summation has been computed. The second expression in Formula 1 represents uncomputed summation and can be bounded. For example, an upperbound of the second expression is $maxw[d_i] \times \sum_{t \in (d_i \cap d_j - \chi)} w_{j,t}$. If the sum of the first expression and the bound of the second expression is less than τ, these two vectors are for sure dissimilar and further computations can be omitted.

Similar formulas are discussed in [4, 18, 7] for computation pruning. Also, some of these techniques, [7] for example, can also be applied in a pre-processing step to eliminate lonely documents with no other vectors to be similar to. While we do leverage such dynamic filtering techniques, our focus is to develop an efficient partitioning method for static filtering and parallelism exploitation.

The work in [22] is a parallel algorithm for Jaccard-based similarity computation which filter vectors by assigning them to the same partition if they share a feature. Comparisons are then conducted within each partition exclusively. Such a mapping is more effective for vectors with short lengths. Different long documents that share many features can collide in multiple partitions and this can result in a significant repetition of computation.

Locality Sensitive Hashing (LSH) can be used to derive approximated APSS results [12, 11]. However, LSH's recall is quite low even though precision is fairly high. Besides, a related study [2] shows that exact algorithms can still deliver performance competitive to LSH when inverted indexing and computation filtering methods are applied [8, 23]. Thus we focus on exact APSS in this paper. However, some application-specific preprocessing techniques can be applied to reduce comparison complexity with an approximation. For example, remove stop-words or extremely high frequent features [16]. During our experiments, datasets are pre-processed with such techniques.

3. PARTITION-BASED APSS

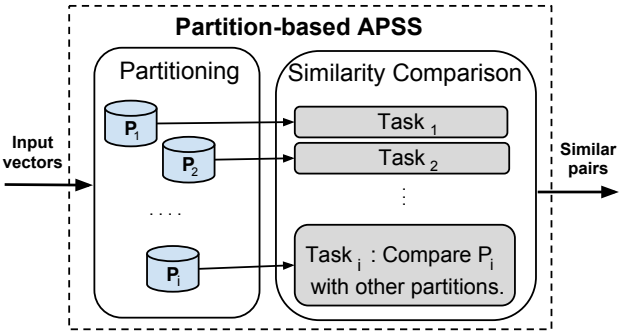


Figure 2: Two-step processing flow of PSS.

In this section, we discuss the overall design of PSS. We assume that PSS runs on a cluster of nodes with multiple CPU cores and can read data from and write data to a distributed file system (DFS) such as Hadoop DFS. The two steps of our

algorithm are depicted in Figure 2. We assume that an input dataset is stored in their natural vector-oriented format on the disk storage. For example, each crawled web page is stored as a set of text words (i.e. features) that it contains. Feature-oriented format (statically stored as inverted index) is less preferable since the maintenance is hard as pages get deleted or their contents get updated periodically.

In the design of our algorithm, we use the following strategies to make it scalable.

- First, unlike previous exact APSS approaches with MapReduce, we run a mapper-only scheme with no reducers. In our approach, the dataset is divided into a set of partitions. Each partition is then assigned to a task to be responsible for computing its similarity with another partition if it has a chance to find similar pairs between those two partitions. We adopt this approach because there are enough task-level parallelism and this simple parallelism management avoids I/O and communication overhead among mappers and reducers. This is further described in Section 5.1.
- Second, during the partitioning process, we mark the dissimilarity relationship among those partitions. In this way, communications and comparisons between dissimilar partitions are avoided. The challenge here is to derive an efficient partitioning mechanism that exposes enough parallelism among the partitions. We also exploit the symmetric nature of pairwise comparison and distribute comparisons evenly among tasks whenever possible. We discuss partitioning and circular computation distribution in Section 4.
- Finally, another consideration is the choice of data structures used in each task. In Section 5, we show that hybrid indexing that partially uses forward indexing, can achieve good performance.

Task$_i$:

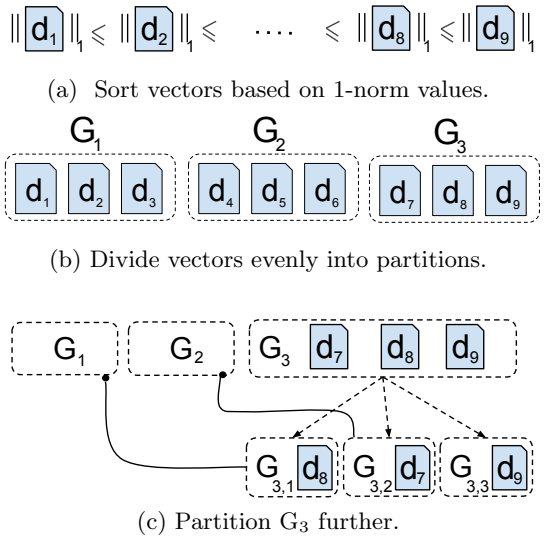
1 Read the assigned partition of s vectors into area $\mathcal{S}$ and compute similarity among them;
2 **repeat**
3 | Read b vectors from another partition into area $\mathcal{B}$;
4 | Compare vectors in $\mathcal{S}$ with vectors in $\mathcal{B}$;
5 | Write comparison results to DFS;
6 **until** *all input vectors in* $\mathcal{D}$ *are compared.*;

Figure 3: High level description of a map task in PSS.

The pseudo-code in Figure 3 defines the basic flow of parallel map tasks in a vector-centric approach. The memory used by each task is divided into three areas. Area $\mathcal{S}$ hosts the assigned partition with an average of s vectors, stored in an inverted index format. Area $\mathcal{B}$ stores b vectors fetched from another partition at each comparison step and stored in forward index format. Area $\mathcal{C}$ stores temporary accumulators holding similarity scores.

4. PARTITIONING AND LOAD BALANCING

This section describes an efficient algorithm that derives a mechanism to partition vectors such that dissimilar vectors are assigned to different partitions. This partitioning algorithm allows each task in our framework to completely avoid fetching partitions from DFS which are dissimilar to the partition it owns. We also discuss how to exploit the symmetric nature of comparison to avoid further overheads while assigning partition computations evenly.



(a) Sort vectors based on 1-norm values.

(b) Divide vectors evenly into partitions.

(c) Partition G_3 further.

Figure 4: Example of static partitioning using 9 vectors.

4.1 Static Partitioning

To determine that two vectors cannot be similar without explicitly computing the product of their features, the following formula based on Formula 1 is used .

$$Sim(d_i, d_j) \leq min(maxw[d_i]||d_j||_1, maxw[d_j]||d_i||_1) \quad (2)$$

If the upper-bound is less than τ, such document pair cannot be similar and comparison between such pairs can be avoided. This formula shows the relationship between two vectors, the challenge is to find a partitioning of vectors quickly without involving a quadratic complexity.

Our algorithm first sorts all vectors based on their 1-norm value (ie.$\|\cdot\|_1$). Without loss of generality, assume that

$$\|d_1\|_1 \leq \|d_2\|_1 \leq \cdots \leq \|d_n\|_1 \ .$$

Given a vector d_j, we find another vector d_i in the above sorted list where:

$$\|d_i\|_1 \leq \frac{\tau}{maxw[d_j]}. \quad (3)$$

Vectors d_i and d_j cannot be similar because this implies $Sim(d_i, d_j) \leq maxw[d_j]\|d_i\|_1 \leq \tau$. Moreover, all vectors $d_1, d_2, \cdots d_{i-1}$ cannot be similar to d_j due to the above norm-based sorting. Following this concept, we rapidly find a large number of vectors dissimilar to one vector.

We define τ-$maxw$ ratio of vector d_j as $\frac{\tau}{maxw[d_j]}$. The above analysis shows that a sorted comparison between the 1-norm value and τ-$maxw$ ratio of vectors can facilitate the identification of dissimilar data partitions.

The partitioning procedure is formally described below along with an example illustrated in Figure 4.

- **Step 1**. Sort all vectors by their 1-norm values ($\|\cdot\|_1$) in a non-decreasing order. Divide this ordered list evenly to produces u consecutive groups $G_1, G_2, \cdots, G_u$.

For example, given $d_1, d_2, \cdots, d_9$ in a non-decreasing order of their 1-norm values, shown in Figure 4, the above step produces the following groups:

$$G_1 = \{d_1, d_2, d_3\},\ G_2 = \{d_4, d_5, d_6\},\ G_3 = \{d_7, d_8, d_9\}.$$

- **Step 2**. Partition each group further as follows. For the i-th group G_i, divide its vectors into i disjoint subgroups $G_{i,1}, G_{i,2}, \cdots, G_{i,i}$. With $j < i$, each subgroup $G_{i,j}$ contains all vectors d_x from group G_i satisfying the following inequality:

$$\max_{d_y \in G_j} \|d_y\|_1 \leq \frac{\tau}{maxw[d_x]}.$$

Let $leader(G_j)$ be the maximum vector 1-norm value in group G_j, namely $\max_{d_y \in G_j} \|d_y\|_1$. The above condition can be interpreted as:

  - $G_{i,1}$ contains G_i's vectors whose τ-$maxw$ ratio is in the interval [$leader(G_1)$, $leader(G_2)$);
  - $G_{i,2}$ contains G_i's vectors whose τ-$maxw$ ratio is in the interval [$leader(G_2)$, $leader(G_3)$);
  - $G_{i,i-1}$ contains G_i's vectors whose τ-$maxw$ ratio is greater or equal to $leader(G_{i-1})$.
  - $G_{i,i}$ contains vectors from G_i that are not in $G_{i,1}$, $G_{i,2}$, $\cdots, G_{i,i-1}$.

For the example in Figure 4(c), group G_3 is further divided into:

$$G_{3,1} = \{d_8\}, G_{3,2} = \{d_7\}, G_{3,3} = \{d_9\}.$$

The vectors in G_3 have been compared with the leaders of groups G_1 and G_2 and satisfied:

$$Leader(G_1) = \|d_3\|_1 \quad \leq \frac{\tau}{maxw[d_8]}$$

and

$$Leader(G_2) = \|d_6\|_1 \quad \leq \frac{\tau}{maxw[d_7]}.$$

The *leader* value defines the characteristic of each partition and facilitates the fast partitioning to detect dissimilarity relationship. From the above partitioning algorithm, it is easy to show that the groups derived by the above algorithm satisfy the following property.

PROPOSITION 1. *Given $i > j$, Vectors in $G_{i,j}$ are not similar to ones in any group $G_{k,l}$ where $l \leq k \leq j$.*

This is because as $G_{i,j}$ is not similar to G_j, any leader in groups with index below j has the vector 1-norm value less than $Leader(G_j)$. Thus $G_{i,j}$ is not similar to G_1,G_2, $\cdots, G_{j-1}, G_j$.

Figure 5 illustrates the dissimilarity relationship among the partitioned groups and each edge represents a dissimilar relationship. For example, members of $G_{3,2}$, $G_{4,2}$,$\cdots$, $G_{n,2}$ are not similar to any member in $G_{1,1}$, $G_{2,1}$ or $G_{2,2}$,

The complexity of this partitioning algorithm is $O(n \log n)$. This is because Step 2 of the above algorithm only needs to compare member's τ-$maxw$ ratios with the leaders of each group, $G_1, \cdots G_{i-1}$. Furthermore, it is easy to parallelize this algorithm with a parallel sorting routine that Hadoop provides. The cost of parallel partitioning is relatively small and is less than 3% of the overall execution time of PSS in our experiments.

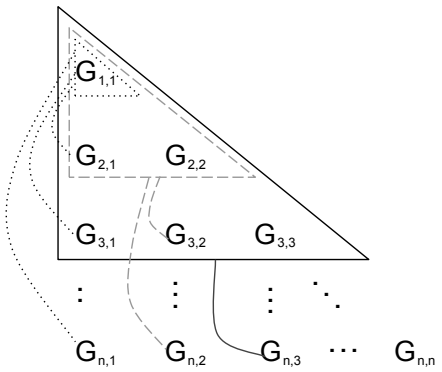


Figure 5: Dissimilarity relationship among partitions.

Step 1 of our static partitioning algorithm uses the 1-norm values to sort vectors. We call it 1-norm guided. Another option is to sort based on the $maxw$ value of each vector, then find a point that fulfills the following inequality:

$$maxw[d_j] \leq \frac{\tau}{\|d_i\|_1}. \tag{4}$$

With this sorting, Step 2 is modified to divide group G_i into i disjoint subgroups $G_{i,1}, G_{i,2}, \cdots, G_{i,i}$ using the following criterion. With $j < i$, each subgroup $G_{i,j}$ contains all vectors d_x from group G_i that satisfies the following:

$$\max_{d_y \in G_j} maxw[d_j] \leq \frac{\tau}{\|d_x\|_1}.$$

We call the above alternative partitioning algorithm as ∞-norm guided. The choice of 1-norm or ∞-norm guided partitioning depends on characteristics of the dataset.

To further utilize the above partitioning effectively, we first eliminate the lonely features which only appear in a single vector in the entire dataset. These features do not participate in any similarity computation. We also investigated the elimination of lonely vectors following the definition of $\hat{d}$ from [7], where $\hat{d} = \{\hat{w}_{i,1}, \hat{w}_{i,2}, \cdots, \hat{w}_{i,m}\}$. Here $\hat{w}_{i,j} = \max_{d_i \in D}\{w_{i,j}\}$, which is the maximum weight for j-th feature among all vectors. Then, $Sim(d_i, d_j) \leq Sim(d_i, \hat{d})$. If any vector satisfies $Sim(d_i, \hat{d}) \leq \tau$, such vector is a lonely vector and cannot be similar to any vector in the entire dataset. In the datasets we have tested, we have not found enough lonely vectors. However, we find it is useful to exploit partition-level lonely vectors as follows.

Once the static partitions have been derived, we further detect dissimilarity relationships among them using the following procedure.

- For each data partition P_i, compute:

$$\hat{d}_i = \{\hat{w}_{i,1}, \hat{w}_{i,2}, \cdots, \hat{w}_{i,m}\}$$

  where $\hat{w}_{i,j} = \max_{d_i \in P_i}\{w_{i,j}\}$ and it is the maximum weight for the j-th feature used in set P_i.

- Two partitions P_i and P_k are dissimilar if $Sim(\hat{d}_i, \hat{d}_k) \leq \tau$.

4.2 Partition-oriented Symmetric Comparison and Load Balancing

As the similarity result of two vectors is symmetric, we only need to compare vector d_i from $\mathcal{S}$ with d_j from $\mathcal{B}$ when $i < j$ in Line 4 of Figure 3. There are two approaches available.

- *Statement-level symmetric condition.* Each task reads vectors from all other partitions. The symmetric condition is applied after vectors are fetched into area $\mathcal{B}$, so some effort for data fetching is wasted. The total read I/O traffic for loading data into $\mathcal{B}$ in all tasks is $O(n^2/s)$. Only half of data fetched are actually used for comparisons, thus effective read I/O ratio is only 50%, which represents a significant waste and unnecessary time delay.
- *Partition-level symmetric condition.* A better solution is that the condition of symmetric comparison is applied at the partition level. Namely d_i and d_j are compared only when $PartitionID(d_i) < PartitionID(d_j)$, where $PartitionID()$ is the partition ID where each vector belongs to. With such a condition, Line 3 of Figure 3 only reads other partitions when their partition IDs are bigger than the local partition assigned to the task. Hence read I/O of a task is 100% effective in the sense that all fetched vectors from other partitions are compared.

The disadvantage of the above partition oriented strategy is that fetching vectors from other partitons is highly skewed among the tasks. Let p be the number of partitions. The first task for Partition 0 needs to read all other $(p-1)$ partitions and has the highest I/O and computation load. The second task reads $(p-2)$ partitions for comparison. I/O volume and computation load decrease monotonically as the task identification number increases. Task for partition $(p-2)$ only compares with 1 partition while task for partition $(p-1)$ does not read and compare any other partition. The above skewed distribution affects the load balance.

Given the above considerations, we devise a a *circular* load assignment applied at the partition-level that still exploits the symmetric comparison condition.

- If p is odd, Task i, which handles partition i, compares with the following partition IDs:

  $$(i+1) \bmod p,\ (i+2) \bmod p,\ \cdots,\ (i+\frac{p-1}{2}) \bmod p.$$

- If p is even, Task i $(0 \le i < p/2)$ compares with the following partition IDs:

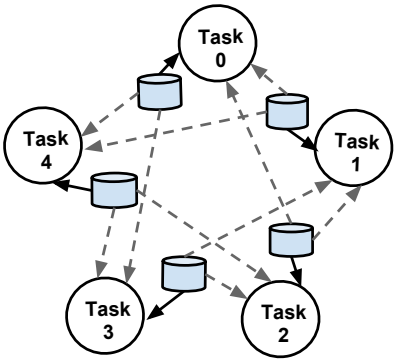
  $$(i+1) \bmod p,\ (i+2) \bmod p,\ \cdots,\ (i+\frac{p}{2}) \bmod p.$$

  Task i $(p/2 \le i < p)$ compares with the following partition IDs:

  $$(i+1) \bmod p,\ (i+2) \bmod p,\ \cdots,\ (i+\frac{p}{2}-1) \bmod p.$$

In this way, every partition is compared with at most $\frac{p}{2}$ other partitions and the computation load is evenly distributed with a difference of at most one partition. Figure 6 illustrates a case with 5 tasks for 5 partitions.

The circular assignment for a pair of partitions is applied only when these two parititons are not marked as dissimilar by the static partitioning algorithm.



Task i	assigned	compared
0	P_0	P_1, P_2
1	P_1	P_2, P_3
2	P_2	P_3, P_4
3	P_3	P_4, P_0
4	P_4	P_0, P_1

Figure 6: Example of partition-oriented circular comparison assignment. Solid arrows indicate local reads, dotted arrows indicate a likely through network reads. Partition-level comparisons are listed in the right table.

5. RUNTIME DATA STRUCTURE AND TRAVERSAL

In this section we discuss the design of the sparse data structure and traversal method for each task. Feature vectors from a web-scale dataset are often very sparse, and this creates challenges in data structure and access design.

One approach is to maintain the assigned vectors in a forward index format. Namely each vector ID is associated with a list of feature IDs and weights, stored in a compact manner. This forward indexing algorithm proceeds as follows: once each task loads its own s vectors, it starts to fetch a block of b vectors from other partitions into $\mathcal{B}$ and compute similarity one by one between these b vectors and the assigned s vectors. This process repeats until all other potentially similar vectors are fetched and compared. To compute the similarity of *each* vector in area $\mathcal{B}$ to the local vectors, the core of the computation is to traverse through the forward index of *all* these s vectors and accumulate the partial similarity score. If the similarity of a vector pair (d_i, d_j) exceeds the predefined threshold τ, a record of the form $(d_i, d_j, score)$ is appended to the output file.

The drawback of the above forward indexing approach is that it takes time to find the common features between these vectors in $\mathcal{S}$ and a vector in $\mathcal{B}$. Even though this scanning can be shortened by using a skipping technique [25], an inverted index approach is more efficient to exploit the sparse data representation and avoid unnecessary similarity comparison when vectors do not share any features [8, 16, 7]. PSS uses a hybrid approach that combines the use of an inverted index with a forward index at the same time, with the objective of taking advantage from both approaches.

5.1 Hybrid Indexing

Figure 7 describes the function of each PSS task with hybrid indexing, called PSS/HI. Each task loads the assigned vectors in a forward index format and then inverts them locally within area $\mathcal{S}$ (Lines 1,2). It then fetches b vectors from other partitions one at time, which are still in forward index format (Line 4). Compared with a fully inverted index approach (such as the one described in Figure 1), our inverted index data is handled locally by each task at runtime instead of an offline processing to invert the data.

Let d_j be a vector fetched from another partition. For each feature t in d_j (Line 8), the inverted index in area $\mathcal{S}$ is

Task:

```
1  Read the assigned s vectors into S;
2  Build inverted index of s vectors and find similarity
   within S;
3  repeat
4  |  Fetch b vectors from another partition into B,
   |  subject to static partitioning and circular
   |  assignment constraints;
5  |  foreach d_j ∈ B do
6  |  |  r_j ← ||d_j||_1;
7  |  |  InitializeArray(score) of size s;
8  |  |  foreach t ∈ d_j and Posting(t) ∈ S do
9  |  |  |  foreach d_i ∈ Posting(t) and
   |  |  |  isCandidate(d_i, d_j) do
10 |  |  |  |  if score[i]+maxw[d_i]×r_j<τ then
11 |  |  |  |  |  (d_i, d_j) is not a candidate;
12 |  |  |  |  else
13 |  |  |  |  |  score[i]=score[i] + w_{i,t}×w_{j,t};
14 |  |  |  r_j ← (r_j − w_{j,t});
15 |  |  for i = 1 to s do
16 |  |  |  if score[i] ≥ τ then
17 |  |  |  |  Write d_i, d_j, score[i];
18 until all required partitions in (D − S) are fetched;
```

Figure 7: Details of each parallel task in PSS/HI.

used to find t's posting localized for the s vectors. Then, the weights of vector d_i in t's posting will contribute a partial score towards $Sim(d_i, d_j)$.

The memory requirement for the temporary partial scores in $\mathcal{C}$ is small, since $\mathcal{C}$ mainly hosts array $score[\,]$ and $maxw[\,]$ with a size proportional to s.

Lines 6,10 and 14, are the implementation of the dynamic filtering over vector d_j from [8] implied from Formula 1. In Line 9, d_i is a candidate being potentially similar to d_j when it is not marked as dissimilar by the dynamic elimination in Line 11. For the tested benchmark data, dynamic elimination integrated with our scheme does not improve the performance for non-binary feature vectors after static filtering, but actually slows it down due to the additional comparison overhead. Only one binary dataset in our experiments has a relatively small improvement in performance. Thus an alternative would be to disable this dynamic filtering.

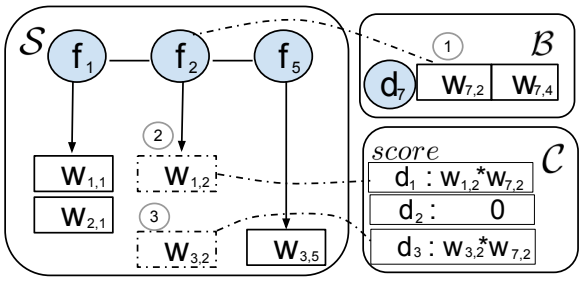


Figure 8: Example of task computation in PSS/HI.

Figure 8 shows one processing example of a task in PSS. When a vector $d_7 = \{w_{7,2}, w_{7,4}\}$ is fetched from another partition and stored in area $\mathcal{B}$, we access the first feature t_2 and traverse its posting in area $\mathcal{S}$. Then each vector weight ($w_{1,2}$ from d_1 and $w_{3,2}$ from d_3 in the example) appearing in t_2's posting is accessed and the partial contribution is accumulated to $score[d_1]$ and $score[d_3]$ in area $\mathcal{C}$.

5.2 Cost Comparison

Here, we analyze briefly the worst-case processing time for a task in PSS using the hybrid indexing and compare it with the forward index approach. By worst-case we mean, none of the computations is dynamically filtered, but every partition needs to be compared with R vectors from other partitions. This simplifies the formula analysis next. As the I/O behavior of both methods is the same, we can safely ignore this part of the comparison and focus on the computational aspect between the two approaches.

In hybrid indexing, there is a small overhead for index inversion of the s vectors assigned to each task. This portion of overhead is negligible because the inverted index is only built once during partition loading and is reused many times during comparison with other partitions. The computation-intensive part is the inner loop, Line 9 in Figure 7. Assume vectors in $\mathcal{S}$ and $\mathcal{B}$ have an average size of k features. Let l be the cost of looking up the posting of a feature in $\mathcal{S}$ where this feature appeared in a vector d_j from $\mathcal{B}$. Term p_s is the average length of postings visited in $\mathcal{S}$, so it estimates the number of iterations for Line 9.

There are 4 memory accesses with a cost of 4δ (Lines 10, 13), namely fetching data items $score[i]$, $w_{i,t}$, $maxw[d_i]$, and writing to $score[i]$. Other items, such as $w_{j,t}$, r_j, and τ, are constants within this loop and can be pre-loaded into registers. These lines also involve 2 pairs of multiplication and addition bringing in a cost of 4ψ, where ψ is the unit execution cost for multiplication or addition. Let T_{HI} be the computation cost of each task in PSS/HI comparing with R vectors from other partitions. Then computation can be estimated as follows.

$$\mathbf{T}_{\text{HI}} \approx Rk[l + p_s(4\delta + 4\psi)].$$

For forward indexing along with dynamic computation filtering called PSS/FI, its computation cost can be estimated as follows:

$$\mathbf{T}_{\text{FI}} \approx Rk[2s\delta + 4p_s\psi].$$

Considering $s \gg p_s$, $\delta \gg \psi$, and l is insignificant compared to other terms, the cost relative ratio of hybrid and forward indexing can be approximated as:

$$\frac{\mathbf{T}_{\text{HI}}}{\mathbf{T}_{\text{FI}}} \approx \frac{l + 4p_s\delta + 4p_s\psi}{2s\delta + 4p_s\psi} \approx \frac{2p_s}{s}.$$

We have verified the above ratio in the experiments. For example, the $\frac{p_s}{s}$ ratio of the Twitter dataset is about 9.97×10^{-2}. Thus, ratio $\frac{T_{HI}}{T_{FI}}$ is estimated to be about 20% and the actual ratio is about 9%. This shows that the above analysis helps to reveal the advantage of hybrid indexing over forward indexing.

6. PERFORMANCE EVALUATION

This section describes the performance evaluation of the parallel algorithms for similarity search as discussed earlier in the paper. These algorithms are implemented in Java

running with Hadoop MapReduce and tested on a 84-node PC cluster. Each node contains 12 cores (Intel X5650 6-core dual processors) and 24GB of memory. The cluster management provides a batch mode for running jobs submitted through a queue and we were able to use about 20 to 40 machines with a total of 120 to 320 cores. The results reported in this section are an average of three runs.

The evaluation of this section has the following objectives:

- First, we assess the scalability and speedup of PSS/HI using three datasets described next.
- Second, we perform comparative experiments among PSS/HI and the following algorithms:
  - ∗ PSS/FI that uses PSS's filtering when applicable but with forward indexing described in Section 5.
  - ∗ SII which is an inverted indexing implementation with a dynamic computation filtering as discussed in Section 2.
  - ∗ A parallel APSS algorithm from [7], called, SSJ-2R, that uses inverted indexing and dynamic computation filtering.
- Finally, we assess the impact of the proposed static partitioning and circular assignment, and compare 1-norm and ∞-norm guided partitioning.

The datasets used in the experiments are as follows:

- **Twitter dataset**: 19.6 million tweets collected from approximately 2 million unique users. The average number of features per vector is 18.5. The total number of features is 980K and 25% which are lonely features.
- **Clueweb dataset** : 50 million web pages, randomly selected from the Clueweb collection from [3]. The average number of features is 320 per web page with approximately 56% of them are lonely features.
- **Emails dataset**: the raw corpus contained 619,446 messages belonging to 158 users with an average of 757 messages per user. Our preprocessing yielded 516,302 cleaned messages with 107 features on average per message and a total of 448K features.

The datasets are preprocessed to follow a *tf-idf* weighting scheme after filtering stop words and top 0.5% frequent features [16]. We have used threshold τ values 0.9 and 0.7 in the experiments.

6.1 Scalability and Comparative Studies

We first study the parallel time and speedup of PSS/HI when increasing the CPU resource. Figure 9 shows the speedup of PSS/HI in processing Twitter and Clueweb datasets with $\tau = 0.9$. The speedup using q cores is defined as the ratio of parallel time using q cores divided by the serial time when using one core. The performance of PSS/HI scales well as the number of CPU cores increases. With 200 cores, the efficiency for Twitter is about 80% defined as the speedup divided by the number of cores. The efficiency cannot reach 100% because 1) having more cores introduces higher startup cost; 2) having more machines increases the inter-machine communication overhead. The efficiency for the Clueweb dataset is 61%, lower than the 80% efficiency of

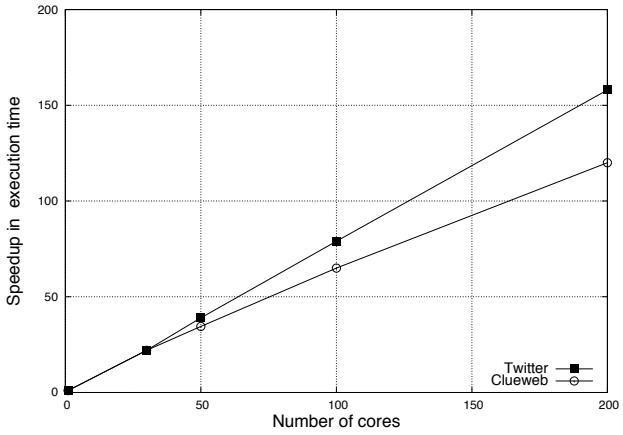


Figure 9: Speedup of PSS/HI while processing 20M Twitter and 50M Clueweb with varied numbers of cores.

Twitter dataset. This is because Clueweb data has higher average number of nonzero features per vector. Also, the tested Clueweb data has much more vectors than Twitter. This results in more map tasks for similarity comparisons, higher startup cost and heavier inter-machine data transfer traffic. Even with such overhead, the overall algorithm performance scales well as the computing resource increases.

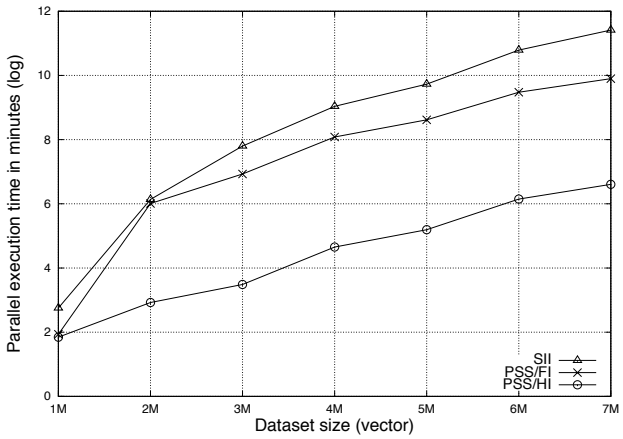
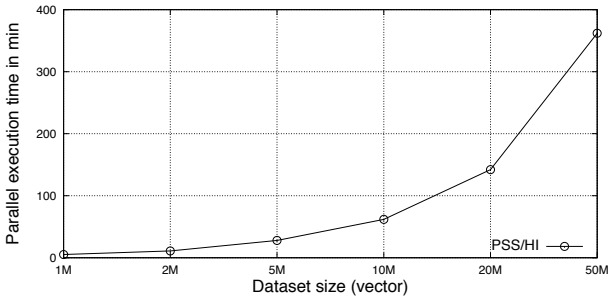


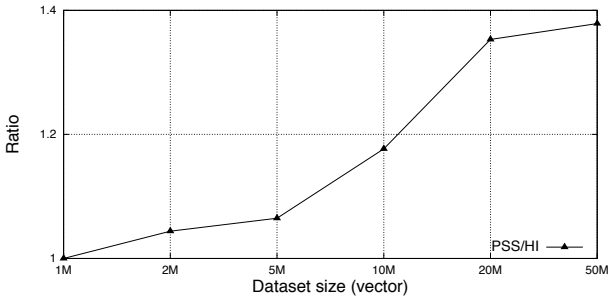
Figure 10: Parallel time on 120 cores of SII, PSS/HI, and PSS/FI over Twitter in *log scale* as data size increases from 1M to 7M with $\tau = 0.9$.

To compare PSS/HI with other algorithms when data size scales up, Figure 10 shows the execution time of PSS/HI, PSS/FI, and SII in log scale when varying the problem size. For a 1 million Twitter dataset, PSS/HI is about 2 times faster than SII while it is similar to PSS/FI's. As data size scales up, we noticed a relatively large gap between PSS/FI and SII. PSS/HI can be up to 25 times faster than SII due to the excessive shuffling between map and reduce tasks. PSS/FI can be up to 10 times slower than STHI because of the slow scanning of common features between vectors in area $\mathcal{S}$ and a vector in area $\mathcal{B}$.

Figure 11(a) shows the parallel time of PSS/HI when dealing with the Clueweb dataset of different sizes using 320 cores. Figure 11(b) shows the relative ratio of the parallel time increase compared to the data size increase, which is defined as:



(a)



(b)

Figure 11: (a) Parallel time of PSS/HI for Clueweb with 320 cores as the dataset scales from 1M to 50M. (b) The growth ratio of time over data size increases using Formula 5.

$$\frac{PT_n}{PT_{baseline}} / \frac{n}{n_{baseline}} \quad (5)$$

where n is the dataset size and PT_n is the corresponding parallel time. In this case, $n_{baseline}$ is 1M and $PT_{baseline}$ is the parallel time for 1M dataset. The curve shows that PSS/HI exhibits a scalable behavior with a near linear increase of parallel time as the data size increases.

Alg.	Time	3M	5M	7M
PSS/HI	Computation	511.63	1,004.35	1,934.73
	I/O	27.16	49.44	68.59
	Total	538.783	1,053.80	2,003.33
PSS/FI	Computation	6,214.01	12,074.40	21,234.70
	I/O	31.60	56.35	69.44
	Total	5,849.01	11,308.7	19,638.0

Table 1: Average processing time in seconds for computations and I/O of each PSS/FI and PSS/HI map task using Twitter data with varying sizes.

Table 1 explains why PSS/HI outperforms PSS/FI by showing the I/O and computation distribution of each PSS/FI or PSS/HI task in processing the Twitter dataset of different sizes. The I/O time of two algorithms is about the same, while the computation time reduction ratio from PSS/FI to PSS/HI is over 90%. PSS/HI outperforms PSS/FI by exploiting the sparsity of inverted index locally at each machine and arranging memory allocation and data traversal more effectively.

We further compare PSS/HI with another inverted index implementation. Table 2 shows the running time of SSJ-2R when processing a smaller Twitter dataset varying from 60K to 100K. The reducer input column shows the number of intermediate results as output of map tasks communicated to the reduce tasks. These numbers are reported by Hadoop. The reducer output column is the number of similar vector pairs produced. This table shows large numbers of intermediate results are generated, compared to the size of final output. As the problem size increases, the amount of data shuffled between map tasks and reduce tasks increases significantly. PSS/HI is much faster than SSJ-2R for these tested cases. We are not able to conduct this direct comparison with a larger dataset as the current SSJ-2R code implementation from [7] may have a size limitation.

Dataset size n	Reducer input	Reducer output	SSJ-2R	PSS/HI
60K	394,123,600	328,778	1,969	158
80K	741,378,156	1,077,014	3,180	216
100K	1,284,741,377	1,428,190	6,180	228

Table 2: Parallel time of SSJ-2R vs. PSS/HI in seconds for processing small Twitter datasets using 10 cores on 5 machines.

6.2 Effectiveness of partitioning and circular load balancing

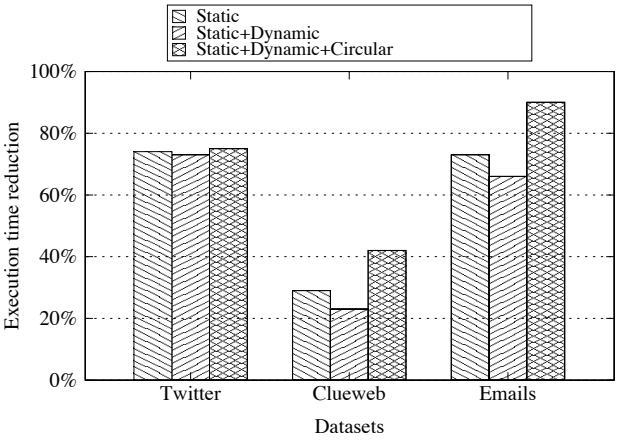


Figure 12: Reduction percentage of execution time when incorporating different optimization techniques. The baseline uses a simple partitioning and statement-level symmetric computation elimination. "Static" implies using static partitioning and filtering over the baseline algorithm. "Static+Dynamic" uses static filtering after partitioning and dynamic filtering over the baseline. "Static+dynamic+Circular" means that all three techniques are applied.

Figure 12 itemizes the percentage of execution time reduction by using the optimization techniques under $\tau = 0.9$. The number of cores allocated is 120 for the Twitter and 10M of Clueweb datasets, and 20 cores for the Email dataset. The baseline is a simpler version of PSS/HI with symmetric comparison filtering. In this baseline version, a comparison of two vectors is skipped if $i < j$, but dynamic computation filtering, static computation filtering with partitioning, or circular assignment is not applied. Each bar represents execution time reduction ratio by using one or more optimization techniques compared to the baseline.

Overall reduction from using all techniques leads to 75%

for Twitter, 42% for Clueweb, and 90% for Emails. Features vectors of these datasets all use non-binary weights.

Static partitioning with dissimilarity detection leads to about 74% reduction for Twitter, about 29% for Clueweb, and about 73% for Emails. Adding the circular assignment contributes an additional 2% for Twitter, 19% for Clueweb, and 14% for Emails.

Dynamic computation filtering after static elimination actually slows down the computation in all these 3 datasets from 1% to 7%. That is because dynamic filtering carries the overhead of extra computation while most dissimilar vectors are already detected by static partitioning. Also even when a vector in $\mathcal{S}$ is detected as dissimilar to another vector from $\mathcal{B}$, the inverted index structure does not change and thus does not reduce the data traversal cost, which is more significant than that of multiplication or addition. Dynamic filtering integrated in our scheme does have some impact for binary vectors.

We have also assessed the impact of the optimization for datasets with binary feature weights. Figure 13 shows the effectiveness of the optimization with a setting similar to Figure 12. Dynamic filtering yields a slight improvement with about 1% reduction for Twitter whose average vector size is relatively short. Static partitioning for dissimilarity detection gains significant reduction while circular assignment is still effective.

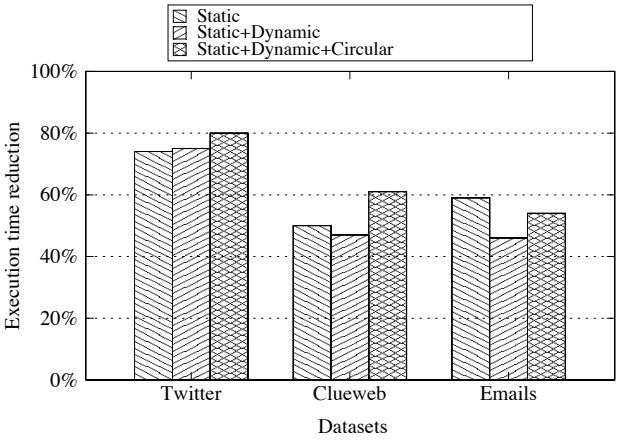


Figure 13: Reduction percentage in execution time under different optimization techniques defined in Figure 12. The data vectors are represented using binary weights.

To materialize the effect of the techniques over the I/O portion of the algorithm, Figure 14 shows the amount of data read I/O reduced by using static partitioning with $\tau = 0.7$ or $\tau = 0.9$ over sampled datasets. This is computed by comparing the total amount of read I/O of PSS/HI algorithm with or without static partitioning. Static partitioning reduces I/O volume by up to 81%.

Figure 15 shows a comparison of total read communication volume reported by Hadoop when using the statement-level and partition-level symmetric comparison discussed in Section 4.2. The results show that the partition-level based one can save up to 50% of the communication volume.

6.3 Partitioning guided by 1-norm and ∞-norm

Table 3 compares the two versions of static partitioning guided by 1-norm or ∞-norm discussed in Section 4.

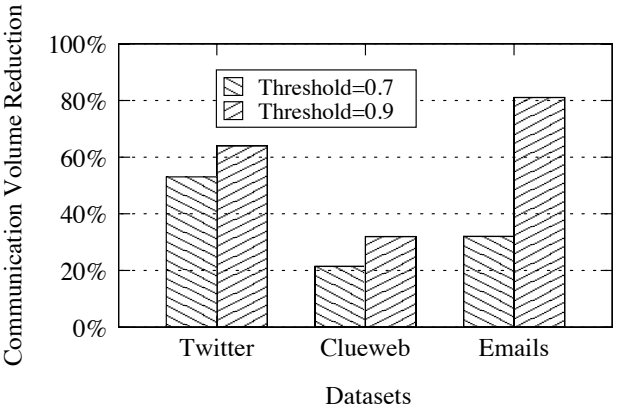


Figure 14: Percentage of the read I/O reduction in bytes, after using the static partitioning under different thresholds.

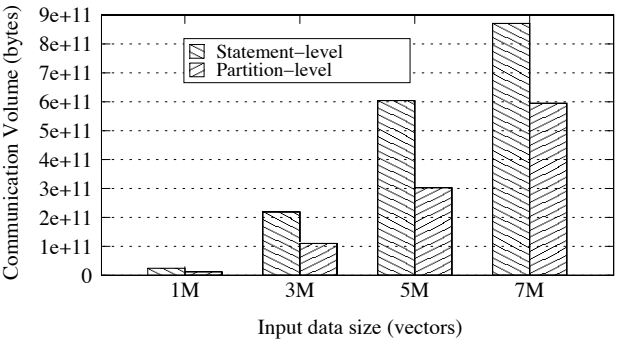


Figure 15: Total communication volume with statement-level symmetric comparison and with partition-level symmetric comparison. The Twitter dataset varies from 1M to 7M vectors.

Datasets	Execution time 1-norm: ∞-norm	Skipped vector pairs 1-norm:∞-norm
Twitter	1.01	1.01
Clueweb	0.88	0.82
Emails	1.09	1.07

Table 3: The second column is relative ratio of execution time between 1-norm ∞-norm guided partitioning. The third column is relative ratio of the number of skipped vector pairs by these two methods.

The table shows the ratio of execution time between these two methods, and also the ratio in terms of the number of skipped vector pairs among the two. The result shows two methods are about the same for Twitter. 1-norm is more effective for Clueweb while the ∞-norm guided method performs better with Emails.

7. CONCLUSIONS

The main contribution of this paper is a two-step partition-based APSS algorithm with 1) an efficient static partitioning and circular assignment for an early removal of unwanted I/O and computations, and 2) a partition-based runtime comparison with a simplified parallelism management and hybrid indexing. Our experiments demonstrate the signifi-

cant performance gain obtained by PSS. For example, PSS is up to about 25 times faster than a parallel approach that exploits parallelism in partial results accumulation for Twitter dataset and is 10 times faster than a forward indexing scheme. The experiments show that the static partitioning with dissimilarity detection significantly improves performance while circular assignment is also effective. The integration of dynamic computation filtering is not effective as expected and we plan to investigate this issue further. We currently assume the dataset to be static, and our future work is to consider incremental computing algorithms for applications where the dataset is updated continuously [24].

Acknowledgment

We thank Alexandra Potapova and Paul Weakliem for their help in data processing and the anonymous referees for their thorough comments. This work is supported by NSF IIS-1118106 and Kuwait University Scholarship. Equipment access is supported by the Center for Scientific Computing at CNSI/MRL under NSF DMR-1121053 and CNS-0960316. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, june 2005.

[2] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, 2006.

[3] Language Technologies Institute at Carnegie Mellon University. The clueweb09 dataset, http://boston.lti.cs.cmu.edu/data/clueweb09.

[4] Amit Awekar and Nagiza F. Samatova. Fast matching for all pairs similarity search. In *International Conference on Web Intelligence and Intelligent Agent Technology Workshop*, 2009.

[5] Amit Awekar and Nagiza F. Samatova. Parallel all pairs similarity search. In *International Conference on Information Knowledge Engineering*, 2010.

[6] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[7] Ranieri Baraglia, Gianmarco De Francisci Morales, and Claudio Lucchese. Document similarity self-join with mapreduce. In *ICDM*, 2010.

[8] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *Proceedings of WWW*, 2007.

[9] Abdur Chowdhury, Ophir Frieder, David A. Grossman, and M. Catherine McCabe. Collection statistics for fast duplicate document detection. *ACM Trans. Inf. Syst.*, 20(2):171–191, 2002.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of OSDI 2004*.

[11] Jimmy Lin Ferhan Ture1, Tamer Elsayed2. No free lunch: Brute force vs. locality-sensitive hashing for cross-lingual pairwise similarity. 2011.

[12] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[13] Hannaneh Hajishirzi, Wen tau Yih, and Aleksander Kolcz. Adaptive near-duplicate detection via similarity learning. In *Proc. of ACM SIGIR*, pages 419–426, 2010.

[14] Nitin Jindal and Bing Liu. Opinion spam and analysis. In *Proceedings of the international conference on Web search and web data mining*, WSDM '08, pages 219–230, 2008.

[15] Aleksander Kolcz, Abdur Chowdhury, and Joshua Alspector. Improved robustness of signature-based near-replica detection via lexicon randomization. In *Proceedings of KDD*, pages 605–610, 2004.

[16] Jimmy Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *SIGIR*, 2009.

[17] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Detectives: detecting coalition hit inflation attacks in advertising networks streams. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 241–250, 2007.

[18] Gianmarco De Francisci Morales, Claudio Lucchese, and Ranieri Baraglia. Scaling out all pairs similarity search with mapreduce. In *8th Workshop on LargeScale Distributed Systems for Information Retrieval (2010)*, 2010.

[19] Mehran Sahami and Timothy D. Heilman. A web-based kernel function for measuring the similarity of short text snippets. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 377–386, 2006.

[20] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[21] Narayanan Shivakumar and Hector Garcia-Molina. Building a scalable and accurate copy detection mechanism. In *Proceedings of the first ACM international conference on Digital libraries*, DL '96, pages 160–168, 1996.

[22] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, 2010.

[23] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08. ACM, 2008.

[24] Shanzhong Zhu, Alexandra Potapova, Maha Alabduljalil, Xin Liu, and Tao Yang. Clustering and load balancing optimization for redundant content removal. In *Proc. of WWW 2012 (21th international conference on World Wide Web), Industry Track*, 2012.

[25] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), July 2006.