Partitioned Similarity Search with Cache-Conscious Data Traversal

Xun Tang, Yelp Inc. and University of California at Santa Barbara
Maha Alabduljalil, Kuwait University
Xin Jin, University of California at Santa Barbara
Tao Yang, University of California at Santa Barbara

All pairs similarity search (APSS) is used in many web search and data mining applications. previous work has used techniques such as comparison filtering, inverted indexing, and parallel accumulation of partial results. However, shuffling intermediate results can incur significant communication overhead as data scales up. This paper studies a scalable two-phase approach called Partition-based Similarity Search (PSS). The first phase is to partition the data and group vectors that are potentially similar. The second phase is to run a set of tasks where each task compares a partition of vectors with other candidate partitions. Due to data sparsity and the presence of memory hierarchy, accessing feature vectors during the partition comparison phase incurs significant overhead. This paper introduces a cache-conscious design for data layout and traversal to reduce access time through size-controlled data splitting and vector coalescing, and it provides an analysis to guide the choice of optimization parameters. The evaluation results show that for the tested datasets, the proposed approach can lead to an early elimination of unnecessary I/O and data communication while sustaining parallel efficiency with one order of magnitude of performance improvement and it can also be integrated with LSH for approximated APSS.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*clustering, search process*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*distributed systems*

Additional Key Words and Phrases: All-pairs similarity search; memory hierarchy; partitioning; data traversal

1. INTRODUCTION

All Pairs Similarity Search (APSS) [Bayardo et al. 2007], which identifies similar objects among a given dataset, has many important applications. For example, collaborative filtering and recommendations based user or item similarity [Aiolli 2013; Cacheda et al. 2011], search query suggestions with similar results [Sahami and Heilman 2006], plagiarism and spam recognition [Shivakumar and Garcia-Molina 1996; Chowdhury et al. 2002; Kolcz et al. 2004; Jindal and Liu 2008], coalition detection for advertisement frauds [Metwally et al. 2007], clustering [Baeza-Yates and Ribeiro-Neto 1999], and near duplicate detection [Hajishirzi et al. 2010; Zhu et al. 2012]. Conducting similarity search is a time-consuming process because the complexity of a naïve APSS can be quadratic to the dataset size. As big data computing with hundreds of millions of objects such as web mining, improvement in efficiency can have a significant impact to speed up discovery and offer more rich options under the same computing constraint.

Previous research on expediting the process has used filtering methods and inverted indexing to eliminate unnecessary computations[Bayardo et al. 2007; Xiao et al. 2008; Arasu et al. 2006; Anastasiu and Karypis 2014]. 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 to compute and collect similarity results in parallel using an inverted index [Lin 2009; Morales et al. 2010; Baraglia et al. 2010; Metwally and Faloutsos 2012]. Unfortunately, the cost of communicating the intermediate partial results is still excessive and such solutions are un-scalable for larger datasets.

We pursue a design that conducts partition-based similarity search (PSS) in parallel with a simplified parallelism management. We statically group data vectors into partitions such that the dissimilarity of partitions is revealed in an early stage. This static

optimization technique allows an early removal of unwanted comparisons which eliminates a significant amount of unnecessary I/O, memory access and computation. Under this framework, similarity comparisons can be performed through a number of tasks where each of them compares a partition of vectors with other candidate vectors. To expedite computation when calculating the similarity of vectors from two partitions, we have further considered the impact of memory hierarchy on execution time. The main memory access latency can be 10 to 100 times slower than the L1 cache latency. Thus, the unorchestrated slow memory access can significantly impact performance.

We investigate how data traversal and the use of memory layers affect the performance of similarity comparison, and propose a cache-conscious data layout and traversal scheme that reduces the execution time for exact APSS. We propose two algorithms PSS1 and PSS2 to exploit the memory hierarchy explicitly. PSS1 splits the data hosted in the memory of each task to fit into the processor's cache and PSS2 coalesces data traversal based on a length-restricted inverted index. We provide an analytic cost model and identify the parameter values for optimized performance. Contrary to common sense in choosing a large split size, the optimum split size is rather small so that core data can fully reside in the fast cache.

The rest of this paper is organized as follows. Section 2 reviews background and related work. Section 3 gives an overview of PSS. Section 4 presents the static partitioning algorithm. Section 5 discusses the design framework and PSS1 algorithm for cache-aware data splitting. Section 6 analyzes cost model of PSS1 and demonstrates the impact of the parameters on memory access performance. Section 7 presents PSS2, the further optimized algorithm using vector coalescing. Section 8 describes the integration of PSS and Locality-Sensitive Hashing (LSH) for approximated APSS. Section 9 discusses an extension for incremental computing and a revision of PSS for Jaccard and Dice similarity. Section 10 is our experimental evaluation that assesses the benefits of PSS1 and PSS2. Section 11 concludes this paper.

2. BACKGROUND AND RELATED WORK

Following the work in [Bayardo et al. 2007], the APSS problem is defined as follows. Given a set of vectors $d_i = \{w_{i,1}, w_{i,2}, \cdots, w_{i,m}\}$, where each vector contains at most m features and may be normalized to a unit length, the cosine-based similarity between two normalized vectors is computed as:

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

Two vectors d_i, d_j are considered similar if their similarity score exceeds a threshold τ, namely $Sim(d_i, d_j) \geq \tau$. The time complexity of computing APSS is high, especially for a big dataset. Application-specific methods could be applied to reduce the computation complexity. For example, text mining removes stop-words or features with extremely high frequency [Baeza-Yates and Ribeiro-Neto 1999; Lin 2009]. We adopt these methods in the pre-processing step when applicable during our experiments for the tested datasets.

Generally speaking, there are three groups of optimization techniques developed in previous work to accelerate APSS.

— **Dynamic computation filtering**. Partially accumulated similarity scores can be monitored at runtime and dissimilar document pairs can be detected dynamically based on the given similarity threshold, without complete derivation of final similarity scores [Bayardo et al. 2007; Xiao et al. 2008; Morales et al. 2010; Anastasiu and Karypis 2014].

— **Similarity-based grouping and mapping**. The search scope for similarity can be reduced when potentially similar vectors are placed in the same group. One can use an inverted index [Xiao et al. 2008; Lin 2009; Morales et al. 2010; Metwally and Faloutsos 2012] developed for information retrieval [Baeza-Yates and Ribeiro-Neto 1999]. This approach identifies vectors that share at least one feature as potentially similar, so data traversal can be avoided for certain vector pairs. Similarly, the work in [Vernica et al. 2010] maps feature-sharing vectors to the same group for group-wise parallel computation. This technique is more suitable for vectors with low sharing pattern, otherwise it suffers from excessive redundant computation among groups. Locality-Sensitive Hashing (LSH) is a technique that groups similar vectors into one bucket with approximation [Gionis et al. 1999; Ture et al. 2011]. This approach has a trade-off between precision and recall, and can introduce redundant computation when multiple hash functions are used. The work in [Satuluri and Parthasarathy 2012] studies the use of LSH with approximated APSS, but has not considered the integration with the exact APSS. A study [Arasu et al. 2006] shows that exact comparison algorithms can deliver performance competitive to LSH when computation filtering is applied.
— **Parallelism management**. The similarity score of vectors involves a large number of partial result addition and the work in [Lin 2009; Baraglia et al. 2010; Metwally and Faloutsos 2012] leverages the inverted index to exploit parallelism. The MapReduce framework called V-SMART-JOIN in [Metwally and Faloutsos 2012] builds the inverted index and generates the candidate pairs of vectors that share the same features. Then V-SMART-JOIN aggregates the partial scores from all candidate pairs. Another study [Wang et al. 2013] introduces a division scheme to improve load balance for dense APSS problems using multiple rounds of MapReduce computation.

Cache optimization for computationally intensive applications is studied in the context of general database query processing [Shatdal et al. 1994; Boncz et al. 1999]. In particular, the problem of hash join in a main memory DBMS has attracted much attention. Radix-cluster [Manegold et al. 2002] is a partitioning algorithm that utilized an analytic model to incorporate memory access costs when executing hash-join operations. These techniques are typically applied to the database join using one attribute. The computation studied in this paper focuses on similarity search that involves many common features among vector pairs. Cache optimization for computationally intensive applications is also studied in the context of matrix-based scientific computing [Duff et al. 2002; Dongarra et al. 1990; Vuduc et al. 2002; Shen et al. 2000]. The work in [Sundaram et al. 2013] studies cache-aware optimization in dynamic LSH index update and query processing, but has not addressed on the exact APSS. Motivated by these studies, we investigate the opportunities of cache-conscious optimization targeting the APSS problem.

Our work focuses on exact APSS and is complementary to the existing work. For example, dynamic filtering [Bayardo et al. 2007; Xiao et al. 2008] is used in our runtime partition-wise comparison. Inverted index [Xiao et al. 2008; Lin 2009; Metwally and Faloutsos 2012] guides through the runtime data traversal in our scheme. LSH approximation or other mapping [Gionis et al. 1999] can be used to map vectors first to groups, and then static partitioning and fast comparison among partitions studied in this paper can be further applied within each group. The work in [Sundaram et al. 2013] studies cache-aware optimization with dynamic LSH index update and query processing, but has not addressed the problem of computing exact APSS.

Our previous work studied the use of static partitioning in [Alabduljalil et al. 2013b] and cache-aware optimization in [Alabduljalil et al. 2013a]. This paper presents a more general framework of Partition-based Similarity Search with comprehensive analysis

on its memory hierarchy performance, especially when feature-based vector coalescing is used. This paper also studies the integration with LSH to tackle the approximated APSS problem, an extension for incremental vector update, and an extension for applying other similarity metrics. This paper briefly discusses the optimization of load balancing for mapping partitioned tasks and more optimization techniques can be found in [Tang et al. 2014].

3. OVERVIEW OF PARTITION-BASED SIMILARITY SEARCH

In this section, we describe the design considerations and processing flow of the PSS framework. The basic ideas are summarized as follows.

— A large amount of comparisons can be eliminated statically. An earlier detection of such unnecessary computations avoids unwanted data I/O and memory access. To facilitate such a process, we statically group data vectors into partitions such that dissimilar vectors are placed to different partitions as much as possible. This partitioning identifies dissimilar vectors without explicitly computing the product of their features.
— The previous work [Lin 2009; Morales et al. 2010; Baraglia et al. 2010; Metwally and Faloutsos 2012] for parallel APSS exploits parallelism in accumulating the partial score $w_{i,t} \times w_{j,t}$ based on Formula (1) between two vectors d_i and d_j. For example, three weight multiplications in a dot product expression $0.1*0.3+0.3*0.2+0.7*0.5$ can run in parallel, and the partial scores can be accumulated gradually. We call this approach as parallel score accumulation. In addition to parallelism in score accumulation for the dot product of each vector pair, there is a huge number of document pairs which can be computed independently. The above parallel score accumulation excessively exploits parallelism and can cause a large amount of unnecessary management overhead. A partition-based approach has an opportunity to expose sufficient coarse-grain parallelism while incurring less overhead than parallel score accumulation.
— We further consider the performance impact of memory hierarchy when calculating the similarity of vectors from two partitions. Memory access can be $\sim$100x slower than L1 cache and un-orchestrated slow memory access incurs significant cost, dominating the entire computation. Extreme data sparsity creates additional challenges in data structure design and memory access optimization. For example in the webpage dataset we have tested, each page has about 320 features and the number of nonzero features of each vector divided by the total number of features is only around 0.0023%.
— For APSS applications where LSH approximation is necessary, we need to investigate how this framework can be used together with LSH. We will also consider an extension for different similarity measures and incremental computing.

The framework for PSS consists of two phases. The first phase divides the dataset into a set of partitions. During this process, the dissimilarity among partitions is identified so that unnecessary data I/O and comparisons among them are avoided. The second phase assigns a partition to each task at runtime and each task compares this partition with other potentially similar partitions. Figure 1 depicts the whole process. We will present the static detection of dissimilar vectors and the partitioning algorithm in Section 4 and the runtime execution of PSS tasks and optimization with memory hierarchy in Section 5. Then we discuss cache-aware optimization and other extensions in the rest of this paper.

Once the dataset is separated into v partitions, v independent tasks are composed. Each task is responsible for a partition and compares this partition with all potentially similar partitions. We assume that the assigned partition for each task fits in the memory of one machine as the data partitioning can be adjusted to satisfy such

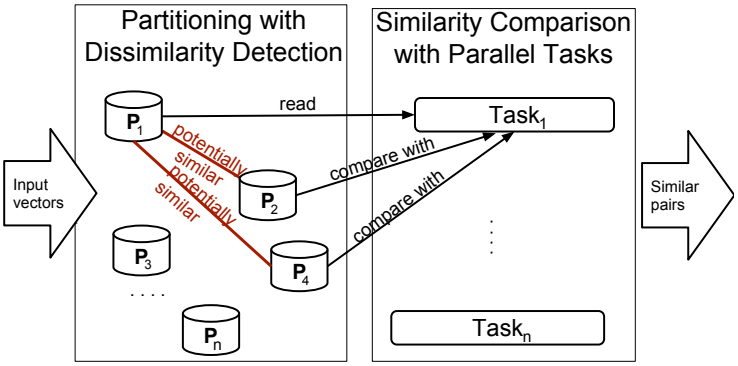


Fig. 1: Illustration of partition-based similarity search.

an assumption. Other partitions to be compared with may not fit in the remaining memory and need to be fetched gradually from a local or remote storage. Task can be executed in a simple multi-core server or in a computer cluster with a distributed file system such as Hadoop, where tasks can seamlessly fetch data from the file system without worrying about the physical locations of data. Each task loads the assigned partition and produces an inverted index to be used during the partition-wise comparison. It then fetches a number of vectors from potentially similar partitions and compares them with the local partition. This process is repeated until all candidate partitions are compared.

4. STATIC PARTITIONING AND DISSIMILARITY DETECTION

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 dissimilar partitions to be compared with the partition it owns.

To identify more dissimilar vectors without explicitly computing the product of their features, we use Hölder's inequality to bound the similarity of two vectors:

$$Sim(d_i, d_j) \leq \|d_i\|_r \|d_j\|_s$$

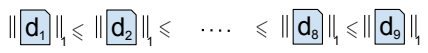
where $\frac{1}{r} + \frac{1}{s} = 1$. $\|\cdot\|_r$ and $\|\cdot\|_s$ are r-norm and s-norm values. r-norm is defined as

$$\|d_i\|_r = (\sum_t |w_{i,t}|^r)^{1/r}.$$

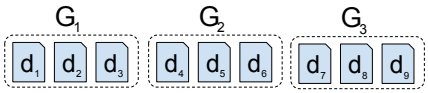
With $r = 1$, $s = \infty$, the inequality becomes $Sim(d_i, d_j) \leq \|d_i\|_1 \|d_j\|_\infty$. If the above similarity upper-bound is less than τ, such vectors are not similar and comparison between them can be avoided.

Although the formula shows the relationship between two vectors, the challenge is to find a quick partitioning of vectors without involving quadratic complexity. Although the above formula shows the relationship between two vectors, the challenge is to find a partitioning of vectors efficiently without involving quadratic complexity. We discuss the basic idea of a partitioning algorithm based on Hölder's inequality as follows. It first sorts all vectors based on their r-norm value (ie.$\|\cdot\|_r$). Without loss of generality, assume that

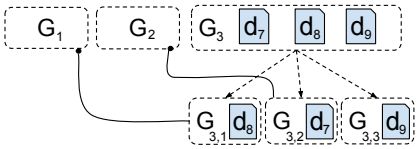
$$\|d_1\|_r \leq \|d_2\|_r \leq \cdots \leq \|d_n\|_r \ .$$



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



(b) Divide vectors evenly into partitions.



(c) Partition G_3 further.

Fig. 2: Example of static partitioning using 9 vectors when $r = 1$ and $s = \infty$.

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

$$\|d_i\|_r < \frac{\tau}{\|d_j\|_s}. \tag{2}$$

Then the above pair of vectors d_i and d_j cannot be similar because Expression (2) implies that $Sim(d_i, d_j) \leq \|d_i\|_r \|d_j\|_s < \tau$. 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 idea, we rapidly find a large number of vectors dissimilar to one vector.

We define τ-s ratio of vector d_j as $\frac{\tau}{\|d_j\|_s}$ The above analysis shows that a sorted comparison between the r-norm value and τ-s 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 2.

— **Step 1**. Sort all vectors by their r-norm values ($\|\cdot\|_r$) in a non-decreasing order. Divide this ordered list evenly to produce 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 r-norm values shown in Figure 2 with $r = 1$, 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:

$$\max_{d_y \in G_j} \|d_y\|_r < \frac{\tau}{\|d_x\|_s}.$$

Let $leader(G_j)$ be the maximum vector 1-norm value in group G_j, namely $\max_{d_y \in G_j} \|d_y\|_r$. The above condition can be interpreted as:
— $G_{i,1}$ contains G_i's vectors whose τ-s ratio is in the interval $[leader(G_1), leader(G_2))$;
— $G_{i,2}$ contains G_i's vectors whose τ-s ratio is in the interval $[leader(G_2), leader(G_3))$;
— $G_{i,i-1}$ contains G_i's vectors whose τ-s 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 2(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:

$$Leader(G_1) = \|d_3\|_1 < \frac{\tau}{\|d_8\|_\infty}.$$

and

$$Leader(G_2) = \|d_6\|_1 < \frac{\tau}{\|d_7\|_\infty}.$$

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, $\cdots, G_{j-1}, G_j$.

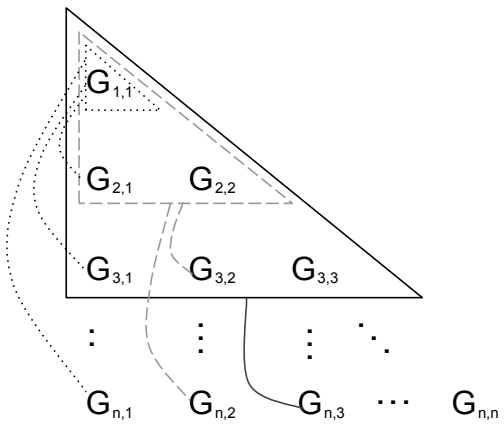


Fig. 3: Dissimilarity relationship among partitions.

Figure 3 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 a member's τ-∞ 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 experiment benchmarks.

To facilitate load balancing in the later phases, we aim at creating more evenly-sized partitions at the dissimilarity detection phase. One way is to divide the large group into smaller partitions. The weakness of this approach is that it introduces more potential similarity edges among these partitions, hence the similarity graph produced becomes denser, more communication and I/O overhead will be incurred during runtime. Another method uses a non-uniform group size. For example, let the size of group G_k be proportional to the index value k, following the fact that the number of subgroups in G_k is k in our algorithm. The main weakness of the second approach is that less dissimilarity relationships are detected as the top layers become much smaller.

We adopt a hierarchical partitioning that identifies large subgroups, detects dissimilar vectors inside these subgroups, and recursively divides them using the above partitioning procedure. The recursion stops for a subgroup when it reaches a partition size threshold. Each partition inherits the dissimilar relationship from its original subgroup.

Data preparation and partition postprocessing. To utilize the above partitioning more 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 [Baraglia et al. 2010] based on [Bayardo et al. 2007; Anastasiu and Karypis 2014]. Here $\hat{d} = \{\hat{w}_{i,1}, \hat{w}_{i,2}, \cdots, \hat{w}_{i,m}\}$ where $\hat{w}_{i,j} = \max_{d_i \in D}\{w_{i,j}\}$ is the maximum weight for the j-th feature among all vectors. Then we have $Sim(d_i, d_j) \leq Sim(d_i, \hat{d})$. Any vector that satisfies $Sim(d_i, \hat{d}) \leq \tau$ is defined as a lonely vector and cannot be similar to any other vector in the entire dataset. In the datasets we 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 $\hat{w}_{i,j} = \max_{d_i \in P_i}\{w_{i,j}\}$ is the maximum weight for the j-th feature used in set P_i.

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

5. RUNTIME COMPARISON OF PARTITIONS

After a dataset is divided into v partitions, there are v corresponding independent tasks. Each PSS task is responsible for a partition and compares this partition with all potentially similar partitions. In this section, we give an overview of a runtime framework for partition-based APSS and then present the caching optimization strategy in detail. We will discuss the load balancing and symmetry of vector comparison in Section 9.

5.1. Runtime Framework and Data Layout

Figure 4 depicts a task in PSS interacting with a CPU core with multiple levels of cache. Two to three cache levels are typical in today's Intel or AMD architecture [Levinthal 2009; Kanter 2010]. We assume that the assigned partition A fits the memory of one machine as the data partitioning can be adjusted to satisfy such an assumption. However, vectors in the other partition (denoted with parameter O) will most likely exceed memory size and need to be fetched gradually from a local or remote storage. In a computer cluster with distributed file system like Hadoop, a task

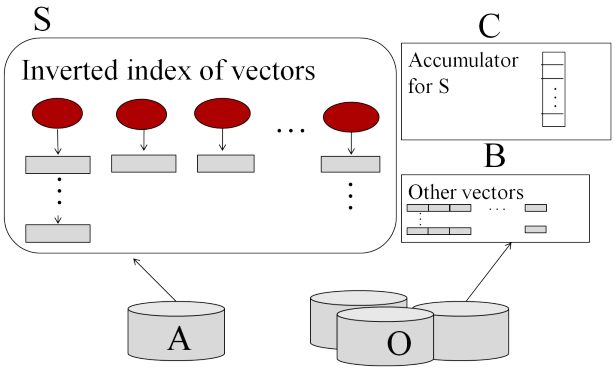


Fig. 4: A PSS task compares the assigned partition A with other partitions O.

ALGORITHM 1: PSSTask(A, O)

Input: Partition A assigned to the task, and another candidate partition O.
Output: Similar pairs from A and O, and their corresponding similarity scores.

```
1  Read all vectors from assigned partition A into S;
2  build inverted index of these vectors and store in S;
3  repeat
4      fetch a set of vectors from O into B;
5      for d_j ∈ B do
6          PSSCompare(S, d_j);
7      end
8  until all vectors in O are fetched;
```

can seamlessly fetch data from the file system without worrying about the machine location of data.

The memory used by each task has three areas, as illustrated in Figure 4.

(1) Area S: hosts the assigned partition A.
(2) Area B: stores a block of vectors fetched from another candidate partition O.
(3) Area C: stores intermediate results temporarily.

Algorithm 1 and Function PSSCompare describe a PSS task. Each task loads the assigned vectors, whose data structure is in forward index format, into area S. Namely, each vector consists of an ID along with a list of feature IDs and their corresponding weights, stored in a compact manner. After loading the assigned vectors, the task inverts them locally within area S. It then fetches a number of vectors from O, in forward index format, and place them into area B.

Let d_j be the vector fetched from O to be processed (Line 5 in Algorithm 1). For each feature t in d_j, PSS uses the inverted index in area S to find the localized t's posting (Line 3 in Function PSSCompare). Then weights of vector d_i from t's posting and d_j contribute a partial score towards the final similarity score between d_j and d_i. After all the features of d_j are processed, the similarity scores between d_j and the vectors in S are validated (Line 13 in Function PSSCompare) and only those that exceed the threshold are written to disk. The dissimilarity of vector d_i in S with d_j can be marked (Line 7 in Algorithm 1) by using a negative value for $score[i]$. Array $||d||_\infty[\,]$ contains the ∞-norm value of vector d_i. The $score[\,]$ vector is also used for dynamic elimination, where a negative value of $score[i]$ indicates d_i marked as a non-candidate.

Function PSSCompare(S, d_j)

```
1   Initialize array score of size |S| with zeros;
2   r_j = ||d_j||_1;
3   for t ∈ d_j and posting(t) ∈ S do
4       for d_i ∈ posting(t) and d_i is a candidate do
5           score[i]=score[i]+w_{i,t}×w_{j,t} ;
6           if (score[i]+||d_i||_∞×r_j<τ) then
7               mark d_i as non-candidate;
8           end
9       end
10      r_j = r_j − w_{j,t};
11  end
12  for i = 1 to |S| do
13      if score[i] ≥ τ then
14          write (d_i, d_j, score[i]);
15      end
16  end
```

5.2. Cache-Conscious Data Splitting

When dealing with a large dataset, the number of vectors in each partition is high. Having a large number of vectors increase the benefits of inverted indexing. But a new problem emerges: the accessed areas S or C may not fit in the fast cache. If that is the case, temporal locality is not exploited, meaning the second access of the same element during any computation will be a cache miss. As we will show in the next section, large inverted index leads to frequent slow memory accesses and a significant increase in execution time. Since fast accesses of areas S, B and C are equally important in the core computation (Lines 5 and 6 in Function PSSCompare), one idea is to let area C fit in L1 cache by explicitly dividing vectors of the assigned partition (S) into a set of splits and have the task focus on one split at a time.

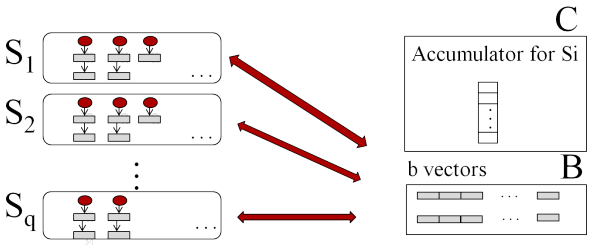


Fig. 5: A partition in area S is further divided into multiple splits for each PSS1 task.

Figure 5 and 6 illustrate this cache-conscious data splitting idea. The corresponding algorithm called PSS1 is shown in Algorithm 2. First, it divides the hosted vectors in S into q splits, each with s vectors. Each split S_i is of size s. PSS1 then executes q comparison sub-tasks. Each sub-task compares vectors from S_i with a vector d_j in B. The access in area C is localized such that array $score[\,]$ and $||d||_\infty[\,]$ can fully fit in L1 cache. This constraint improves temporal locality of data elements for area C and reduces the access time by an order of magnitude. As a result, the core computation speeds up.

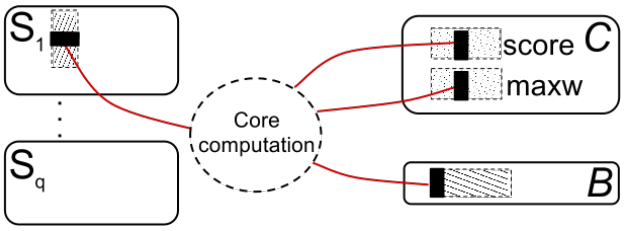


Fig. 6: Core computation in PSS1 and its interaction with data items. Four data items are involved in the core computation. The striped areas indicate cache coverage.

ALGORITHM 2: PSS1Task(A, O)

Input: Partition A assigned to the task, and another candidate partition O.
Output: Similar pairs and their corresponding similarity scores.

```
1   Read and divide A into q splits;
2   repeat
3       build an inverted index for each split S_i and store in S;
4       repeat
5           fetch a set of vectors from O into B;
6           for d_j ∈ B do
7               for S_i ∈ S do
8                   PSSCompare(S_i, d_j);
9               end
10          end
11      until all vectors in O are fetched;
12  until all splits in A are processed;
```

The question is, how do we determine the number of vectors in each split (s value) to optimize cache utilization? This is discussed next.

6. CACHE PERFORMANCE AND COST ANALYSIS OF PSS1

We model the total execution time of each PSS1 task and analyze how memory hierarchy affects the running time. This analysis facilitates the identification of optimized parameter setting. Table I describes the parameters used in our analysis. They represent the characteristics of the given dataset, algorithm variables, and the system setting.

6.1. Task Execution Time

The total execution time for each task includes two parts: I/O and computation. I/O cost occurs for loading the assigned vectors A, fetching other potentially similar vectors, and writing similarity pairs to disk storage. Notice that in fetching other vectors for comparison, the algorithm always fetches a block of vectors to amortize the start-up cost of I/O. For the datasets we have used, read I/O takes about 2% of total cost while write I/O takes about 10-15%. Since I/O cost is the same for the baseline PSS and our proposed schemes, we do not model it in this paper.

For each split, the computation time contains a small overhead for the index inversion of its s vectors. Because the inverted index is built once and reused every time a partition is loaded, this part of computation becomes negligible and the comparison time with other vectors dominates. The core part is computationally intensive. Following notations defined in Table I, h is the cost of looking up the posting of a feature

Table I: Notations

Dataset	
$w_{d,t}$	Weight of feature t in vector d
τ	Similarity threshold
k	Average number of nonzero features in d
Algorithm	
S, B, C	Memory usage for each task
n	Number of vectors to compare per task ($\|O\|$)
s	Avg. number of vectors for each split in S
b	Number of vectors fetched and coalesced in B
p_s, p_b	Average posting length in inverted index of each S_i or B touched
S_i	A split in area S divided by PSS1
q	Number of splits in S
h	Cost for t-posting lookup in table
$m_j(X)$	Miss ratio in level j cache for area X
$D_j(X)$	Number of misses in level j cache for area X
D_j	Total number of access misses in level j cache
δ_{total}	Cost of accessing the hierarchical memory
Infrastructure	
l	Cache line size
f	Prefetch factor
e_s, e_b, e_c	Element size in S, B, C respectively
$\delta_1, \delta_2, \delta_3, \delta_{mem}$	Latency when accessing L1, L2, L3 or memory
ψ	Cost of addition and multiplication

appeared in a vector in B. p_s denotes the average length of postings visited in S_i (only when a common feature exists), so p_s estimates the number of iterations for Line 3 in Function PSSCompare. Furthermore, there are four memory accesses in Line 5 and 6, regarding data items $score[i]$, $w_{i,t}$, $w_{j,t}$, and $||d_i||_\infty$. Other items, such as r_j, and τ, are constants within this loop and can be pre-loaded into registers. The write back of $score[i]$ is not counted due to the asymmetric write back mechanism adopted. The dynamic checking of whether d_i is a candidate or not (Line 7) is an access to $score[\,]$ vector as well (negative indicates non-candidate), and is not modeled separately. There are two pairs of multiplication and addition involved (one in Line 5 and one in Line 6) bringing in a cost of 2ψ. For simplicity of the formula, we model the worst case where none of the computations are dynamically filtered.

For a large dataset, the cost of self-comparison within the same partition for each task is negligible compared to the cost of comparisons with other vectors in O. The execution time of PSS1 task (Algorithm 2) can be approximately modeled as follows.

$$\text{Time} = q\left[\, nk(\overbrace{h}^{lookup} + \overbrace{p_s \times 2\psi}^{multiply+add}) + \overbrace{\delta_{total}}^{traverse\ S,B,C} \,\right]. \tag{3}$$

As s increases, q decreases and the cost of inverted index lookup may be amortized. In the core computation, p_s increases as s increases. More importantly, the running time can be dominated by δ_{total} which is the data access cost due to cache or memory latency. The data access cost is affected by s because of the presence of memory hierarchy. We investigate how to determine the optimal s value to minimize the overall cost in the following subsection.

Table II: Cases of cache miss ratios for split S_i and area C in PSS1 at different cache levels. Column 2, 4, and 6 are the cache miss ratio $m_j(S_i)$ for accessing data in S_i. Column 3, 5, and 7 are the cache miss ratio $m_j(C)$ for accessing data in C.

Case	m_1		m_2		m_3		Description
	S_i	C	S_i	C	S_i	C	
(1)	$\max(\frac{1}{p_s}, \frac{e_s}{fl})$	0	0	0	0	0	C fits L1; S_i does not fit L1, but fits L2.
(2)	$\max(\frac{1}{p_s}, \frac{e_s}{fl})$	$\frac{e_c}{fl}$	0	0	0	0	S_i and C do not fit L1, but fit L2.
(3)	$\max(\frac{1}{p_s}, \frac{e_s}{fl})$	$\frac{e_c}{fl}$	1	0	0	0	C does not fit L1, but fits L2; S_i does not fit L2 but fits L3.
(4)	$\max(\frac{1}{p_s}, \frac{e_s}{fl})$	$\frac{e_c}{fl}$	1	1	0	0	S_i and C do not fit L2, but fit L3.
(5)	$\max(\frac{1}{p_s}, \frac{e_s}{fl})$	$\frac{e_c}{fl}$	1	1	1	0	C does not fit L2 but fits L3; S_i does not fit L3.
(6)	$\max(\frac{1}{p_s}, \frac{e_s}{fl})$	$\frac{e_c}{fl}$	1	1	1	1	S_i and C do not fit L3.

6.2. Memory and Cache Accesses of PSS1

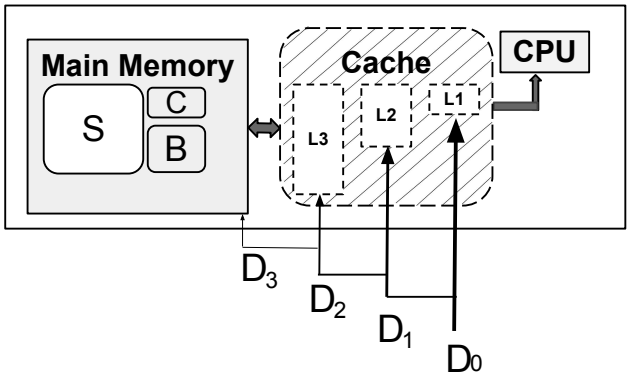


Fig. 7: Data access misses for three-layer cache hierarchy, where $D_{j-1} \geq D_j$, j=1, 2, 3.

Here we estimate the cost of accessing data in S_i, B, and C. As illustrated in Figure 7, D_0 is defined as the total number of data accesses in performing COMPARE(S_i, d_j) in Algorithm 2. D_j is defined as the total number of data access misses in cache level j. δ_i is the access time at cache level i. δ_{mem} is the memory access time.

$$\delta_{total} = (D_0 - D_1)\delta_1 + (D_1 - D_2)\delta_2 + (D_2 - D_3)\delta_3 + D_3\delta_{mem}. \tag{4}$$

To conduct the computation in Lines 5 and 6 of Function PSSCompare, the program needs to access weights from S_i, weights from B, and $score[\,]$ and $||d||_\infty[\,]$ from C. We model these accesses separately then add them together as follows:

$$D_0 = D_0(S_i) + D_0(B) + D_0(C) = \overbrace{nkp_s}^{S_i} + \overbrace{nkp_s}^{B} + \overbrace{2nkp_s}^{C}. \tag{5}$$

Define $D_j(X)$ as the total number of data accesses missed in cache level j for accessing area X. $m_j(X)$ is the cache miss ratio to access data for area X in cache level j.

$$\begin{aligned} D_j =& D_j(S_i) + D_j(B) + D_j(C) \\ =& D_{j-1}(S_i) * m_j(S_i) + D_{j-1}(B) * m_j(B) + D_{j-1}(C) * m_j(C). \end{aligned} \tag{6}$$

Table II lists six cases of miss ratio values $m_j(S_i)$ and $m_j(C)$ at different cache levels j. The miss ratio for B is not listed and is considered close to 0 assuming it is small enough to fit in L1 cache after warm-up. That is true for our tested datasets. For a dataset with long vectors and B cannot fit in L1, there is a small overhead to fetch it partially from L2 to L1. Such overhead is negligible due to the relative small size of B, compared to S_i and C.

A cache miss triggers the loading of a cache line from next level. We assume the cost of a cold cache miss during initial cache warm-up is negligible and the cache replacement policy is LRU-based. Thus the cache miss ratio for consecutive access of a vector of elements is $\frac{1}{l/e}$ where l is the cache line size and e is the size of each element in bytes. We assume that cache lines are the same in all cache levels for simplicity, which matches the current Intel and AMD architecture.

The computer system prefetches a few cache lines in advance, in anticipation of using consecutive memory regions [Levinthal 2009; Kanter 2010]. Also, an element might be re-visited before it is evicted, where the second cache miss is saved. As an example, a popular feature in the inverted index of S_i might be hit again before replacement. We model both factors to the effective prefetch factor f. Let f be the effective prefetch factor for S_i, and e_s be the element size for S_i. The cache miss ratio for accessing S_i is adjusted as $\frac{e_s}{fl}$.

We further explain the cases listed in Table II.

— In Case (1), s is small. C can fit in L1 cache. Thus after initial data loading, its corresponding cache miss ratios $m_1(C_1)$, $m_2(C_1)$, and $m_3(C_1)$ are close to 0. Then $m_1(S_i) = \frac{e_s}{fl}$, and $m_2(S_i)$ and $m_3(S_i)$ are approximately 0 since each split can fit in L2 (but not L1). In this case, s is too small, the benefit of using the inverted index does not outweigh the overhead of the inverted-index constructions and dynamic look-up.

— In Case (2), S_i and C can fit in L2 cache (but not L1). $m_1(S_i) = \frac{e_s}{fl}$, and $m_1(C) = \frac{e_c}{fl}$. $m_2(S_i)$ and $m_3(S_i)$ are approximately 0. Thus δ_{total} is:

$$\begin{aligned}\delta_{total} &= (D_0 - D_1)\delta_1 + D_1\delta_2 \\ &= \left[nkp_s(1 - \max(\frac{1}{p_s}, \frac{e_s}{fl})) + nkp_s + 2nkp_s(1 - \frac{e_c}{fl}) \right] \delta_1 \\ &+ \left[nkp_s \max(\frac{1}{p_s}, \frac{e_s}{fl}) + 2nkp_s\frac{e_c}{fl} \right] \delta_2.\end{aligned} \tag{7}$$

Hence task time is

$$Time = q\left[nk(h + p_s 2\psi) + nkp_s\Big(4\delta_1 + (\max(\frac{1}{p_s}, \frac{e_s}{fl}) + \frac{2e_c}{fl})(\delta_2 - \delta_1)\Big)\right].$$

— As s becomes large in Case (3) to Case (6), S_i and C cannot fit in L2 nor L3, and they need to be fetched periodically from memory if not L3.

A comparison of data access time between PSS1 and PSS.. For a large dataset, Case (6) reflects the behavior of PSS as each partition tends to hold a large number of vectors. PSS1 performs the best with the Case (2) setting and thus we compare the reduction of total data cost from Case 6 to Case (2) in Table II. The D_0 and D_1 values of two cases are the same while $D_2 = D_3 = 0$ in Case (2) and $D_3 = D_2 = D_1$ in Case (6).

$$\frac{\delta_{total}(PSS)}{\delta_{total}(PSS1)} = \frac{(D_0 - D_1)\delta_1 + D_1\delta_{mem}}{(D_0 - D_1)\delta_1 + D_1\delta_2} = 1 + \frac{\delta_{mem} - \delta_2}{(\frac{D_0}{D_1} - 1)\delta_1 + \delta_2}.$$

$\frac{D_1}{D_0}$ represents L1 miss ratio and in practice, it exceeds 10%. On the other hand, δ_{mem} is two orders of magnitude slower than L1 access latency δ_1. Thus data access of PSS1 can be 5 to 10 time faster than that of PSS.

Optimal choice of s. From the above analysis, a larger s value tends to lead to the worst performance. We illustrate the s value for the optimal case on an AMD architecture. For the AMD Bulldozer 8-core CPU architecture (FX-8120) tested in our experiments, L1 cache is of size 16KB for each core. L2 cache is of size 2MB shared by 2 cores and L3 cache is of size 8MB shared by 8 cores. Thus 1MB on average for each core. Other parameters are: $\delta_m = 64.52ns$, $\delta_3 = 24.19ns$,$\delta_2 = 3.23ns$, $\delta_1 = 0.65ns$, $l = 64$ bytes. We estimate $\psi = 0.16ns$, $h = 10ns$, $p_s = 10\%s$, $f = 4$ based on the results from our micro benchmark. The minimum task time occurs in Case (2) when S_i and C can fit in L2 cache, but not L1. Thus the constraint based on the L2 cache size can be expressed as

$$s \times k \times e_s + 2s \times e_c \leq 1MB.$$

While satisfying the above condition, split size s is chosen as large as possible to reduce q value. For Twitter data, k is 18, e_s is 28 bytes, and e_c is 4 bytes. Thus the optimal s is around 2K.

To support the above analysis, Figure 8 shows the actual data-access-to-computation ratio collected from our experiment using Twitter dataset. when s varies from 100 to 25,000. We measure the ratio of the data access time (including the inverted index lookup) over the computation time. This ratio captures the data access overhead paid to perform comparison computation and the smaller the value is, the better. For Twitter benchmark, the above ratio is 8 for optimum case, while it increases to over 25 for Case (3) and Case (4) where more frequent access to L3 cache is required. It shows that by selecting the optimal s value based on our cost function, we are able to reduce the data-access-to-computation ratio from 25 to 8.

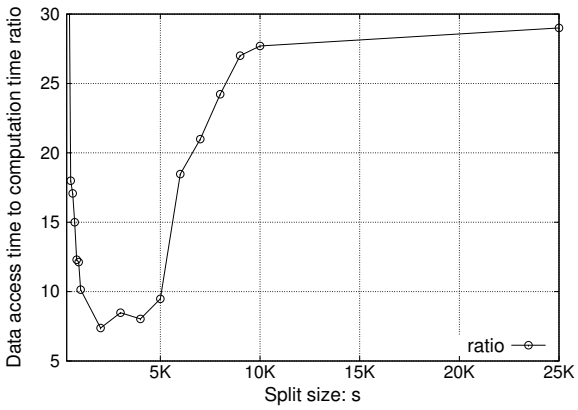


Fig. 8: Y axis is the ratio of actual data access time to computation time for Twitter data observed in our experiments.

ALGORITHM 3: PSS2Task(A, O).

Input: Partition A assigned to the task, and other candidate partitions O.
Output: Similar pairs and their corresponding similarity score.

```
1  Read A and divide it into q splits of s vectors each;
2  build an inverted index for each split S_i;
3  repeat
4      fetch b vectors from O and build inverted index in B;
5      for S_i ∈ S do
6          PSS2Compare(S_i, B);
7      end
8  until all vectors in O are compared;
```

7. FEATURE-BASED VECTOR COALESCING

In PSS1, every time a feature weight from area S_i is loaded to L1 cache, its value is multiplied by a weight from a vector in B. L1 cache usage for S_i is mainly for spatial locality. Namely fetching one or few cache lines for S_i to avoid future L1 cache miss when consecutive data is accessed. Temporal locality is not exploited much, because the same element is unlikely to be accessed again before being evicted, especially for L1 cache due to its small size. Another way to understand this weakness is that the number of times that an element in L1 loaded for S_i can be used to multiply a weight in B is low before this element of S_i is evicted out from L1 cache. PSS2 is proposed to exploit temporal locality and adjust data layout and traversal in B in order to increase L1 cache reuse ratio for S_i.

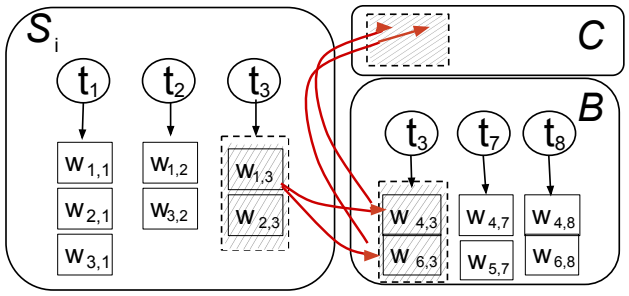


Fig. 9: Example of data traversal in PSS2. Five data items are involved in the core computation. The striped area indicates coverage of cache line.

7.1. Design of PSS2

Figure 9 illustrates the data traversal pattern of PSS2 with $b = 3$. There is one common feature t_3 that appears in both S_i and B. The posting of t_3 in S_i is $\{w_{1,3}, w_{2,3}\}$ and each iteration of PPS2 uses one element from this list, and multiplies it with elements in the corresponding posting of B which is $\{w_{4,3}, w_{6,3}\}$. Thus every L1 cache loading for S_i can benefit two multiplications with weights in B in this example. In comparison, every L1 loading of weights for S_i in PSS1 can only benefit one multiplication.

Algorithm 3 and Function PSS2Compare describe a PSS2 task. The key distinctions from a PSS1 task are as follows.

Function PSS2Compare(S_i, B).

```
1   Initialize array score of size s × b with zeros;
2   for j = 1 to b do
3       r[j] = ||d_j||_1;
4   end
5   for feature t appears in both B and S do
6       for d_i ∈ posting(t) in S do
7           for d_j ∈ posting(t) in B and d_i is a candidate do
8               score[i][j]=score[i][j]+w_{i,t}×w_{j,t};
9               if (score[i][j]+||d_i||_∞×r[j]<τ) then
10                  mark pair d_i and d_j as non-candidate;
11              end
12          end
13      end
14      for d_j ∈ posting(t) in B do
15          r[j] = r[j] − w_{j,t};
16      end
17  end
18  for i = 1 to s do
19      for j = 1 to b do
20          if score[i][j] ≥ τ then
21              write (d_i, d_j, score[i][j]);
22          end
23      end
24  end
```

— Once an element in S_i is loaded to L1 cache, we compare it with b vectors from B at a time. Namely group S_i from S is compared with b vectors in B (Line 6 in Algorithm 3).
— We coalesce b vectors in B and build an inverted index from these b vectors. The comparison between S_i and b vectors in B is done by intersecting postings of common features in B and S_i (Line 5 in Procedure PSS2Compare).
— The above approach also benefits the amortization of inverted index lookup cost. In PSS1, every term posting lookup for S_i can only benefit multiplication with one element in B. In PSS2, every look up can potentially benefit multiple elements because of vector coalescing. Thus PSS2 exploits temporal locality of data in S_i better than PSS1.

Compared with PSS1, PSS2 compares S_i with not one, but b vectors in B at a time. The partial result accumulator is expanded as well, from a one-dimensional array $score[\,]$ (of length s) to a two-dimensional array $score[\,][\,]$ of length $s \times b$. This expansion in space allocation, together with the coalescing effect aforementioned, implies that the cache utilization of PSS2 is affected by the choice of s, as well as b.

In this section, we explain in detail why the parameter choice affects the cache utilization, how the parameter choice changes the cache miss ratios by example cases, and eventually generalize the cases in a cache analytic model. For simplicity of presentation, the analysis is applied to PSS2 without dynamic elimination (line 6 and line 7 in Function PSSCompare).

7.2. Parameter Choices for Optimal Cache Utilization

From the analysis for PSS1, s cannot be too small in order to exploit the spatial locality of data in S_i. Now we examine the choice of b as the number of vectors fetched and stored in $\mathcal{B}$.

— We first discuss the benefits of having a large value for b. The primary gain of PSS2 compared to PSS1 is to exploit the temporal locality of data from S_i by coalescing b vectors in area B. Let p_b be the average number of vectors sharing a feature. The L1 cache miss ratio of S_i is reduced by p_b from PSS1 to PSS2. Choosing a large b is better as it increases p_b value.
  Also since we build the inverted index for vectors in $\mathcal{B}$ dynamically, the small b value will not bring enough locality benefit to offset the overhead of building the inverted index. Thus b cannot be too small. In general, $\mathcal{B}$ would not fit L1 cache.
— There is a disadvantage to increase b from the cache capacity point of view. Increasing b values expands the size of variables in $\mathcal{B}$ and $\mathcal{C}$. Then $\mathcal{B}$ and $\mathcal{C}$ may not fit L2 cache anymore.
  Another consideration is that vectors in $\mathcal{B}$ is sparse as shown in our experiment section (Figure 20) and as a result, a large b value does not linearly increase p_b value.

From cache analysis for PSS1, we expect that PSS2 performs best when S_i, $\mathcal{B}$ and $\mathcal{C}$ fit L2 cache but none of them fit L_1 cache.

Since the space of 2D variable $score[\,][\,]$ dominates the usage of area C, the constraint based on the L2 cache size can be expressed as

$$s \times k \times e_s + b \times k \times e_b + s \times b \times e_c \leq \text{ capacity of L2}.$$

For the Twitter dataset and AMD architecture with 1MB L1 cache per core, when b size is around 8 to 32, s value varies from 1,000 to 1,500, the above inequality can hold. The analysis above does not consider the popularity of features among vectors. Since some features are accessed more frequently than the others, we expect that a smaller number of features are shared among vectors but many others are not shared, thus not need to be cached. As a result, the above inequality does not need to include all features in the capacity planning. We expect the optional choice to be slightly larger than the numbers discussed above.

The miss ratios for the above cases are:

$m_1(S_i) = \max(\frac{1}{p_s}, \frac{e_s}{fl}) \cdot \frac{1}{p_b}$, $m_1(B) = \frac{e_b}{fl} \cdot \frac{1}{p_s}$, $m_1(C) = \frac{e_c}{fl} \cdot \frac{1}{p_b}$,
$m_2(S_i) = m_3(S_i) = 0$, $m_2(B) = m_3(B) = 0$, $m_2(C) = m_3(C) = 0$.

We could derive the total access cost of PSS2 in this case as follows

$$\delta_{total}(PSS2) = D_0\delta_1 + D_1(PSS2)(\delta_2 - \delta_1)$$

where $D_1(PSS2)$ denotes the D_1 value when PSS2 is applied and S_i, $\mathcal{B}$ and $\mathcal{C}$ fit L2 cache.

$$D_1(PSS2) = \max(\frac{1}{p_s}, \frac{e_s}{fl})\frac{nkp_s}{p_b} + \frac{e_b nkp_s}{flp_s} + \frac{3e_c nkp_s}{flp_b}.$$

We compare the above result with δ_{total} for PSS1 with Case (2) in Table II.

$$D_1(PSS1) = \max(\frac{1}{p_s}, \frac{e_s}{fl})nkp_s + \frac{2e_c nkp_s}{fl}.$$

where $D_1(PSS1)$ denotes the D_1 value when PSS1 is applied and S_i and C do not fit L1, but fit L2.

Table III: Explanation of case abbreviations in Figure 10.

Algo.	Case	Description
PSS2	pss2-1	Optimal case for PSS2. S_i, $\mathcal{B}$, $\mathcal{C}$ all fit L2.
	pss2-2s	$\mathcal{B}$ and $\mathcal{C}$ fit L2; while S_i does not.
	pss2-2c	S_i and $\mathcal{B}$ fit L2; while $\mathcal{C}$ does not.
	pss2-2sc	$\mathcal{B}$ fits L2; while S_i and $\mathcal{C}$ do not.
	pss2-2bc	S_i fits L2; while $\mathcal{B}$ and $\mathcal{C}$ do not.
	pss2-3sbc	Worst case for PSS2. S_i, $\mathcal{B}$, $\mathcal{C}$ do not fit L3.
PSS1	pss1-1	Optimal case for PSS1. $\mathcal{B}$ fits L1, S_i and $\mathcal{C}$ fit L2.
	pss1-2s	$\mathcal{B}$ fits L1, $\mathcal{C}$ fits L2; while S_i does not.
	pss1-2sc	$\mathcal{B}$ fits L1; while S_i and $\mathcal{C}$ do not fit L2.
	pss1-3s	A poor case for PSS1. $\mathcal{B}$ fits L2; $\mathcal{C}$ fits L3; while S_i does not.

With a relatively large s value, p_s is relatively large. $\max(\frac{1}{p_s}, \frac{e_s}{fl}) = \frac{e_s}{fl}$. Hence,

$$\frac{\delta_{total}(PSS1)}{\delta_{total}(PSS2)} = \frac{D_0\delta_1 + D_1(PSS1)(\delta_2 - \delta_1)}{D_0\delta_1 + D_1(PSS2)(\delta_2 - \delta_1)} \lessapprox \frac{D_1(PSS1)}{D_1(PSS2)} = \frac{e_s + 2e_c}{\frac{e_s}{p_b} + \frac{e_b}{p_s} + \frac{3e_c}{p_b}}. \quad (8)$$

Impact of s and b values on data-access-to-computation ratio. The above analysis assumes that the smallest memory access time is achieve when all three areas fit L2 cache. To validate this, we have further analyzed the cache miss ratio and access time for other cases, and compare their performance in terms the ratio of data access time (including the inverted index lookup time) over the computation time $\frac{\text{Data-access}}{\text{Computation}}$.

Figure 10 plots the data-access-to-computation ratio ratio for the different cases of parameters in PSS1 and PSS2 and PSS2 cases are from Table III in handling the Twitter dataset. This figure confirms that PSS2 reaches the lowest ratio when $\mathcal{S}_\rangle$, $\mathcal{B}$ and $\mathcal{C}$ fit L2 cache, and its data access speed can be upto 14x faster than the others.

Figure 10 also shows the $\frac{\text{Data-access}}{\text{Computation}}$ ratio for optimal case in PSS2 is about 50% lower than the optimal case in PSS1. Such performance gain proves the positive effect of vector coalescing on cache optimization, when p_b value is not too small. It demonstrates the advantage of PSS2 over PSS1 (a significant reduction of the task execution time) by exhibiting good reference locality.

8. INTEGRATION OF LOCALITY SENSITIVE HASHING WITH PSS

We discuss how LSH [Indyk and Motwani 1998; Gionis et al. 1999] can be incorporated with PSS. LSH is an approximate similarity search technique that scales to both large and high-dimensional data sets. Its basic idea is to hash the feature vectors using several hash functions to ensure that similar vectors have much higher probability of collision in buckets than dissimilar vectors. Previous work has applied variants of LSH on cross-language information retrieval problem [Ture et al. 2011] and near duplicate detection [Hajishirzi et al. 2010].

An adaptive approach [Hajishirzi et al. 2010] tunes LSH by concatenating k hash values from each data vector into a single signature for high precision, and by combining matches over l such hashing rounds, each using independent hash functions, for good recall. An illustration is shown in Figure 11. They use $k = 8$ to 256 min-hash functions over $l = 5$ hashing rounds, and each min-hash value takes roughly 20 bits to store. Within each bucket, the Jaccard similarity score of two documents is determined by the number of identical hash values their corresponding signatures share, divided

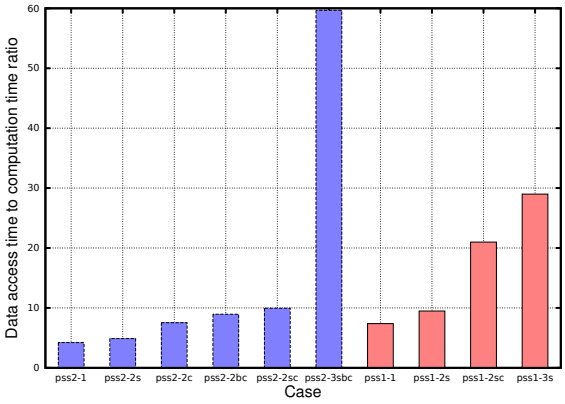


Fig. 10: Y axis is the ratio of actual data access time to computation time for Twitter benchmark observed in our experiments. X axis is the case abbreviation further illustrated in Table III.

by k. The work included in Ivory package [Ture et al. 2011] uses a sliding window mechanism to narrow the scope of similarity comparison on sorted signatures generated by one set of hash functions, and repeats this step for hundreds of rounds. Due to the approximation introduced by bit signatures and the sliding window algorithm, the recall ratio for their method is limited. For example, the recall is 76% with $1,000$-bit signature. If precision is desired, candidates within each LSH bucket could be post-processed by an additional pairwise clustering step by calculating exact similarities to filter out false positives.

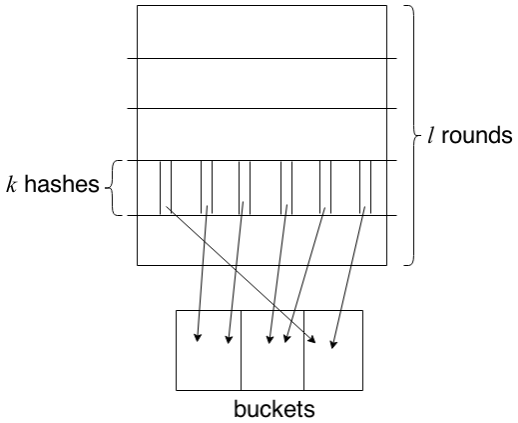


Fig. 11: Illustration of l rounds of LSH, each round generates k hash values.

Since PSS is very fast in dealing with exact APSS, we can apply it for each bucket of vectors mapped by LSH. The exact APSS within different buckets can be done in parallel. Such a pipeline is illustrated in Figure 12. LSH mapping is parallelized over

Table IV: Number of LSH rounds (l) needed to achieve a targeted recall rate for cosine similarity threshold τ with k signature bits.

τ	$recall = 95\%$					$recall = 99\%$	
	$k=3$	$k=5$	$k=7$	$k=9$	$k=11$	$k=3$	$k=5$
0.99	1	1	2	2	2	2	2
0.95	2	3	3	4	4	3	4
0.90	3	4	5	7	8	4	6
0.85	4	6	8	12	17	5	8
0.80	5	8	13	21	34	7	12

the distributed servers in the form of MapReduce jobs, and consists of sub-steps of projection generation, signature generation and bucket generation.

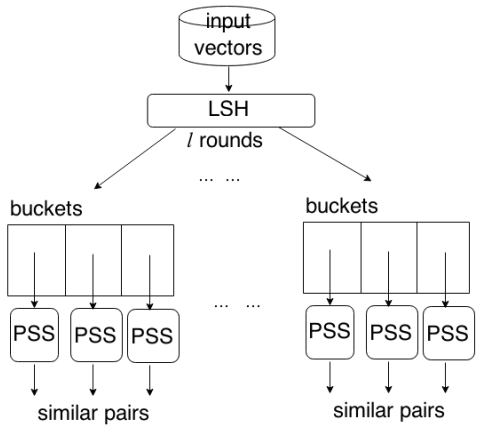


Fig. 12: Integration of LSH with PSS.

We study how the signature bit length k and hashing rounds (l) can be chosen to maximize the performance of integrated LSH with PSS. Let the precision ratio be defined as the percentage of detected similar pairs found by an approximation algorithm is actually similar and the recall ratio be the percentage of similar pairs detected successfully. Since the combined algorithm achieves 100% precision, we opt to use a relatively lower value of k and relatively higher value of l to improve the recall ratio. We compute the required l value to guarantee a recall ratio as follows.

Suppose the probability that two signature bits (at the same position) from two vectors collide equals to the cosine similarity of the two vectors. Given cosine similarity threshold τ, the number of bits for a signature k, and the number of LSH rounds l, the recall ratio is as follows:

$$recall = 1 - (1 - \tau^k)^l.$$

Then l value based on a targeted recall rate is:

$$l = \left\lceil log_{(1-\tau^k)}(1 - recall) \right\rceil.$$

Given various choices of signature bits k, targeted recall ratio, and cosine similarity threshold τ, the corresponding rounds (l) of LSH needed are listed in Table IV.

Because of the approximation introduced in LSH, this combined approach is much faster than pure PSS. If we assume in each round, all vectors are evenly divided among buckets. Considering the overhead of generating LSH signatures and making copies of vectors to different buckets is relatively insignificant, the integrated scheme could reduce the total number of similarity comparisons to $\frac{l}{2^k}$ of the original number. This is because given n vectors, the number of pair-wise similarity comparison is reduced from $\frac{n^2}{2}$ to $\frac{(\frac{n}{2^k})^2}{2}$ in each bucket over a total of $2^k \cdot l$ buckets. However, this ideal speedup ratio may not be reachable because the vector mapping to buckets may not be load balanced and pairs of similar vectors are mapped to multiple buckets, which leads to a redundant computation. The actual speedup obtained in our experiments is reported in Section 10.6.

9. DISCUSSIONS

Task formation for load balancing. The size of these PSS tasks can be highly irregular because static partitioning can produce vector groups with a skewed size distribution. Even runtime task scheduling in a parallel platform such as Hadoop provides load balancing support in executing independent tasks, APSS-specific techniques for optimizing load balancing can yield additional benefits.

One key condition to facilitate a runtime execution scheduler to achieve a good balance is to have a relatively even distribution of overall work load among PSS tasks. We discuss how to improve the evenness of load distribution among tasks in forming comparison tasks. Given two tasks T_i and T_j that own partition P_i and P_j respectively, due to the computation symmetry, only one task needs to compare vectors of P_i with vectors of P_j. An optimization decision needs to select which task should conduct the above partition-wise comparison and this decision affects the load size of T_i or T_j. A poor assignment can lead to a skewed workload distribution among tasks and thus affect the parallel time of the final schedule in the target execution platform.

One simple task formation scheme is to use a circular mapping for balancing task loads and we define it as follows. Let p be the number of partitions produced by static partitioning and data partitions are numbered as $0, 1, \cdots, p-1$. If p is odd, Task T_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 T_i ($0 \leq i < p/2$) compares with the following partition IDs:

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

Task T_i ($p/2 \leq i < p$) 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 with a difference of at most one partition. The above balancing technique is relatively simple and a more complex two-stage assignment algorithm that considers partition size variation is in [Tang et al. 2014].

PPS with-incremental updating. In various applications, some percentage of vector content could be updated periodically. For example, a web search engine constantly crawls the web and there is a percentage of page content change in the searchable database. Twitter users create new tweets and a tweet database is updated every day. How to handle APSS with incrementally updated vectors?

To avoid the computation of similarity scores among vectors from scratch, we focus on the computation of similarity scores between an updated or new vector and an existing vector. One option is to figure out the dissimilarity of new and updated vectors with the existing vectors, merge them with the existing partitions, and then conduct related APSS. We find that this naïve approach takes too much overhead for data redistribution and comparison.

Alternatively,we consider the number of newly added or updated vectors is relatively small, and it is much simpler to compare them with the existing vectors in parallel and then distribute new data to partitions following the dissimilarity relationship. As illustrated in Figure 13 (a). we set aside a *new* partition and every time a vector is updated or a new vector is generated, we append such a vector to the *new* partition. Once this *new* partition grows to a threshold in terms of size or time, we start a parallel PSS job to compare the *new* partition with all the original partitions. This procedure identifies a new set of vector pairs which are similar.

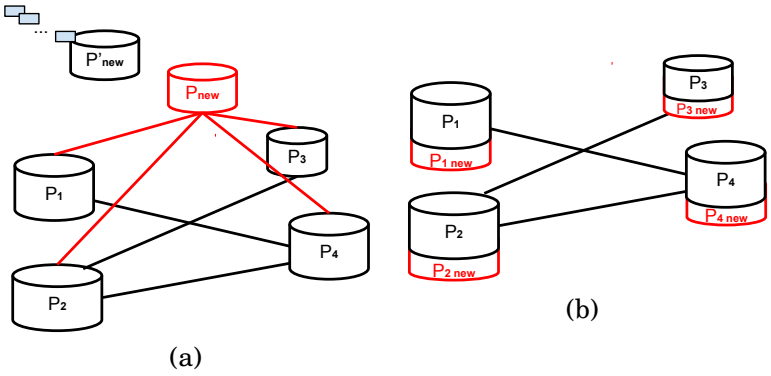


Fig. 13: (a) The pipeline for incremental computing of similarity scores given a set of new or updated vectors. (b) The updated static partitions.

The above approach identifies new similar pairs quickly by going through a one-to-many partition-wise comparison. After that, we need to place each vector of the new partition to the original set of partitions in order to facilitate the future update. Following the notation in Section 4, each vector d_x of the new partition is inserted into group G_i if i is the minimum integer that satisfies $\|d_x\|_r \leq \max_{d_y \in G_i} \|d_y\|_r$. This vector d_x inserted in group G_i is further mapped to subgroup $G_{i,j}$ where j is the maximum integer satisfying $\max_{d_y \in G_j} \|d_y\|_r < \frac{\tau}{\|d_x\|_s}$. Figure 13 (b) illustrates how these existing partitions grow after new vectors are appended based on the aforementioned scheme.

Extension to other similarity measures. Since PSS1 and PSS2 are based on the cosine similarity metric, we discuss an extension to apply our techniques for other two similarity measures with binary vectors.

— **Jaccard similarity**. For binary vectors, the Jaccard similarity is defined as

$$Sim(d_i, d_j) = \frac{\|d_i \cdot d_j\|_1}{\|d_i\|_1 + \|d_j\|_1 - \|d_i \cdot d_j\|_1}.$$

Following the upper bound discussed in [Theobald 2008], it is easy to verify that if one of the following inequalities is true:

$$\|d_i\|_1 < \tau\|d_j\|_1 \text{ or } \|d_j\|_1 < \tau\|d_i\|_1,$$

$Sim(d_i, d_j) < \tau$. The static partitioning algorithm in Section 4 can be revised as follows. It will still sort all vectors by norm $|d|_r$ with $r = 1$. After this sorting and grouping, given the leader value in a group G_i, we can find a vector d_j with largest value j such that $leader(G_i) < \tau|d_j|_1$. Then d_j is dissimilar to any member in G_1, G_2, $\cdots$, G_i. Thus subgroup $G_{i,j}$ is defined as containing these members d_x in G_i satisfying the following inequality.

$$Leader(G_i) < \tau\|d_x\|.$$

For runtime partition comparison, Line 9 of Function PSS2Compare in PSS2 needs to be modified as:

$$score[i][j] + r[j] < \frac{\tau}{1+\tau}(\|d_i\|_1 + \|d_j\|_1).$$

Notice that $score[i][j]$ keeps track of the partially-computed value $\|d_i \cdot d_j\|_1$. Term $score[i][j]$ in Lines 20 and 21 of Function PSS2Compare is replaced with the following Jaccard similarity formula

$$\frac{score[i][j]}{\|d_i\|_1 + \|d_j\|_1 - score[i][j]}.$$

— **Dice similarity**. For binary vectors, the Dice similarity is defined as

$$Sim(d_i, d_j) = \frac{2\|d_i \cdot d_j\|_1}{\|d_i\|_1 + \|d_j\|_1}.$$

It is easy to verify that if one of the following inequalities is true:

$$\|d_i\|_1 < \frac{\tau}{1-\tau}\|d_j\|_1 \text{ or } \|d_j\|_1 < \frac{\tau}{1-\tau}\|d_i\|_1,$$

$Sim(d_i, d_j) < \tau$. Then the static partitioning algorithm in Section 4 can also be modified accordingly after all vectors are sorted by norm $|d|_1$. Namely given the leader value in a group G_i, a vector d_j satisfying $leader(G_i) < \frac{\tau}{2-\tau}|d_j|_1$, is dissimilar to any member in $G_1, G_2, \cdots, G_i$. Thus subgroup $G_{i,j}$ is defined as containing these members d_x in G_i satisfying the following inequality:

$$Leader(G_i) < \frac{\tau}{2-\tau}\|d_x\|.$$

For runtime partition comparison, condition of Line 9 of Function PSS2Compare in PSS2 needs to be modified as:

$$score[i][j] + r[j] < \frac{\tau}{2}(\|d_i\|_1 + \|d_j\|_1).$$

Term $score[i][j]$ in Lines 20 and 21 of Function PSS2Compare is changed with the following Dice formula

$$\frac{2 \cdot score[i][j]}{\|d_i\|_1 + \|d_j\|_1}.$$

10. EXPERIMENTAL EVALUATIONS

We have implemented our algorithms in Java and the parallel solution runs on a Hadoop cluster. The source code and test datasets could be found at a public site https://github.com/ucsb-similarity/pss. Our evaluations have the following objectives:

(1) Demonstrate the benefits of partitioned APSS in parallelism management in a Hadoop cluster and compare it with the alternative parallel solutions [Lin 2009; Baraglia et al. 2010; Metwally and Faloutsos 2012]. Assess the impact of static partitioning.
(2) Compare PSS1 and PSS2 with the baseline PSS using multiple application datasets and illustrate the impact of parameters by examining the cache hit ratios and execution time under different choices.
(3) Report the efficiency and effectiveness of incorporating LSH with PSS, and provide a guideline for the method choices that meet different requirements in dealing with approximated APSS.
(4) Demonstrate the incremental updating with PSS and assess the use of PSS for the Jaccard and Dice metrics.

Datasets. The following five datasets are used.

— Twitter dataset containing 100 million tweets with 18.32 features per tweet on average after pre-processing. Dataset includes 20 million real user tweets and additional 80 million synthetic data generated based on the distribution pattern of the real Twitter data but with different dictionary words.
— ClueWeb dataset containing about 40 million web pages, randomly selected from the ClueWeb09 collection produced by Language Technologies Institute at CMU. The average number of features is 320 per web page. We choose 40M records because it is big enough to illustrate the scalability.
— Yahoo!music dataset (YMusic) used to investigate the song similarity for music recommendation. It contains 1,000,990 users rating 624,961 songs with an average feature vector size 404.5.
— Enron email dataset containing 619,446 messages from the Enron corpus, belonging to 158 users with an average of 757 messages per user. The average number of features is 107 per message.
— Google news (GNews) dataset with over 100K news articles crawled from the web. The average number of features per article is 830.

The datasets are pre-processed to follow the TF-IDF weighting after cleaning and stopword filtering.

Environment setup and metrics. We ran parallel speedup experiments on a cluster of servers with Linux/Hadoop and each node has Intel X5650 6-core 2.66GHz dual processors and 24GB of memory per node. We have also used a cluster of nodes with 4-core AMD Opteron 2218 2.6GHz processors and 8G memory. The cache-conscious experiments were also conducted on 8-core 3.1GHz AMD Bulldozer FX8120 machines. Each AMD FX8120 processor has 16KB of L1 cache per core, 2MB of L2 cache shared by two cores, and 8MB of L3 cache among all eight cores. Each Intel X5650 processor has 32KB of L1 data cache per core, 1.5MB of L2 cache per processor, and 12MB of L3 cache per processor.

To verify our analytic model for optimization of memory hierarchy, we use the results measured by the Linux profiling tool *perf*. Perf collects the performance counters that accumulate hardware events, and helps us understand how the program interacts with a machine's cache hierarchy. For modern machines with three levels of cache, perf collects from the first-level and third-level cache measures.

10.1. Benefits of Partition-based APSS

In this subsection, we demonstrate the parallel performance of PSS when a set of partitioned tasks runs on a Hadoop cluster using only map tasks and compare its per-

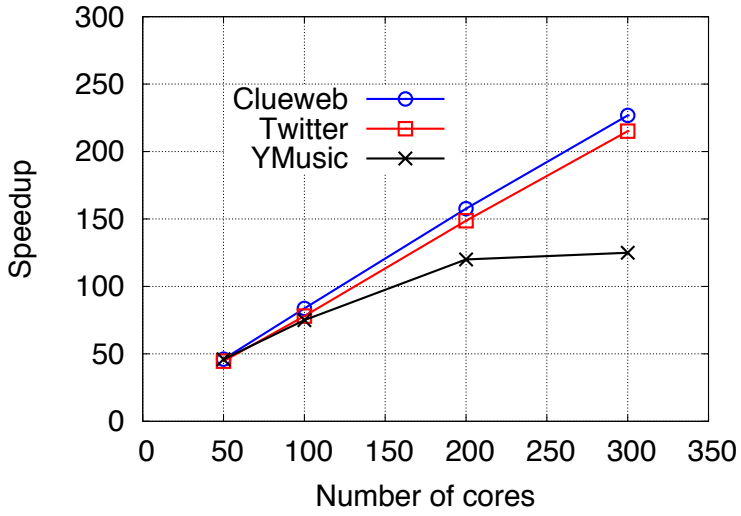


Fig. 14: Speedup for 3 datasets when varying the number of cores used.

formance with the previous work which exploits fine-grain accumulation parallelism with both map and reduce tasks.

Figure 14 shows the speedup for processing 40M ClueWeb dataset and 100M Twitter dataset with similarity threshold 0.8 when varying the number of cores on the Intel cluster. Speedup is defined as the estimated serial time of these tasks divided by the parallel time. We will discuss the estimated serial cost in next paragraph. It should be noted that the time measurement is conducted when all machines are dedicate and there is no machine failure during the execution. The performance of our scheme scales well as the number of CPU cores increases. The efficiency is defined as the speedup divided by the number of cores used. For the two larger datasets, the efficiency is about 83.7% for ClueWeb and 78% for Twitter when 100 cores are used. When running on 300 cores, the efficiency can still reach 75.6% for ClueWeb and 71.7% for Twitter. The decline is most likely caused by the increased I/O and communication overhead among machines in a larger cluster. Efficiency for YMusic is 76.2% with 100 cores and 42.6% with 300 cores. There is no significant reduction of parallel time from 200 cores to 300 cores, remaining about 15 minutes. The problem size of this dataset is not large enough to use more cores for amortizing overhead. Still parallelization shortens search time and that can be important for iterative search experimentation and refinement. Enron email or Gnews dataset are not used in the scalability experiments due to similar reasons.

The serial time of processing Yahoo music is 17.8 hours on Intel when the system is configured to run one map task at a time. The above baseline is the fastest Java implementation that we can optimize after incorporating the work of [Bayardo et al. 2007]. The estimated serial time for Twitter 100M and ClueWeb 40M datasets is 25,438 hours and 45,157 hours respectively. We estimate them because it takes too long to run sequentially. For Twitter 4M, the serial time is 26.7 hours while for Twitter 10M, it takes 254 hours. For ClueWeb 1M, the serial time is 29.3 hours hours while for ClueWeb 4M, it takes 469 hours. The time cost of this algorithm grows approximately in a quadratic manner. We assume the preprocessing of input vectors such as stopword removal is already completed and do not employ any additional preprocessing approximation such as removing features with their frequency exceeding an upper limit [Lin 2009]. Such

preprocessing can further significantly reduce the serial time with approximated results.

To confirm the choice of partition-based search which does not use reduce tasks, we have implemented an alternative MapReduce solution to exploit parallel score accumulation following the work of [Lin 2009; Baraglia et al. 2010; Metwally and Faloutsos 2012] where each mapper computes partial scores and distributes them to reducers for score merging. The performance comparison is presented in Figure 15 The parallel score accumulation is much slower because of the communication overhead incurred in exploiting accumulation parallelism. Even with extensive optimization to filter unnecessary computations, 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. Such communication between mappers and reducers, purely conducted over disk and network transfer, becomes cumbersome after dataset size growing over tens of millions of records. For example, to process 4M Twitter data using 120 cores, parallel score accumulation is 19.7x slower than partition-based similarity search which has much simpler parallelism management and has no shuffling between mappers and reducers. To process 7M Twitter data, parallel score accumulation is 25x slower. The approach of parallel score accumulation is much slower because of the communication overhead incurred in exploiting accumulation parallelism. In comparison, execution of PSS does not involve reduce tasks, there is much less communication cost and there are less chances experiencing the straggler effect.

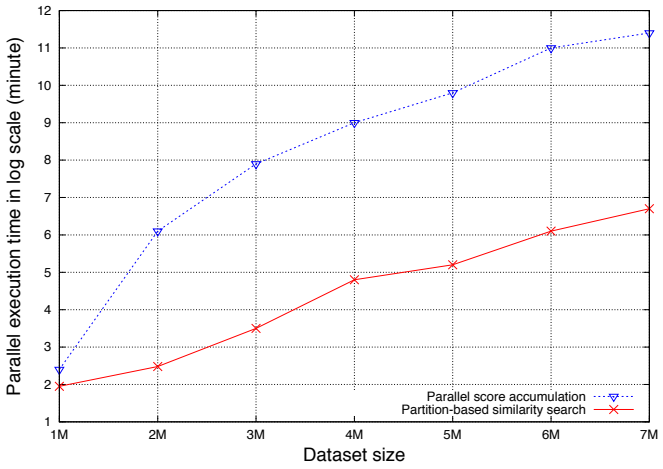


Fig. 15: Parallel time on 120 cores of parallel score accumulation method and partition-based similarity search over Twitter dataset as data size increases from 1M to 7M (τ = 0.9). Time is reported in log scale.

Table V shows that static partitioning, which is parallelized, takes 2.1% to 3% of the total parallel execution time. This table also shows the time distribution in terms of data I/O and CPU usage for similarity comparison. Data I/O is to fetch data and write similarity results in the Hadoop distributed file system. The cost of self-comparison

Dataset	Cores	Static Partitioning	Similarity Comparison		
			Read	Write	CPU
Twitter	100	2.8%	0.9%	11.7%	84.6%
ClueWeb	300	2.1%	1.9%	7.8%	88.2%
YMusic	20	3.0%	2.3%	1.8%	92.9%
Emails	20	1.0%	1.7%	8.7%	88.6%

Table V: Cost of static partitioning and runtime cost distribution of PSS in parallel execution.

among vectors within a partition is included when reporting the actual cost. The result of this table implies that the computation cost in APSS is dominating and hence reducing the computation cost of tasks is critical for overall performance. Later we will show the impact of PSS1 and PSS2 in reducing computing cost.

10.2. Impact of Static Partitioning

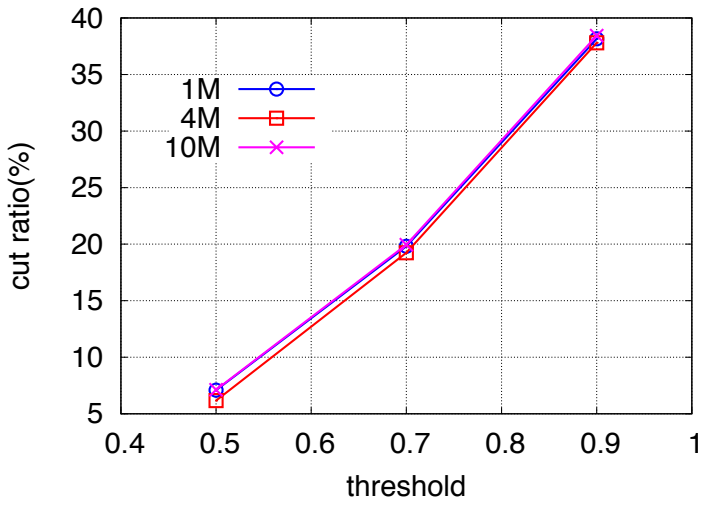


Fig. 16: Percentage of skipped comparison with static partitioning for Twitter 1M, 4M and 10M under different thresholds.

Figure 16 shows the percentage of comparisons skipped using static data partitioning when varying the similarity threshold for Twitter dataset with 1M, 4M, and 10M vectors. The norms discussed in Section 4 are set as $t = 1$ and $s = \infty$. For Twitter 10M, the skipped percentage is 38.44% with $\tau = 0.9$ and drops to 7.12% with $\tau = 0.5$. The reason that a lower similarity threshold detects less dissimilar pairs is because the upper bound $\frac{\tau}{\|d_j\|_s}$ in Formula (2) becomes smaller with a smaller τ value. Thus static filtering is most effective for applications using relatively high similarity thresholds.

Figure 17 shows the impact of choosing different r-norms discussed in Section 4 on the percentage of comparisons skipped using static data partitioning. The similarity threshold is $\tau = 0.8$ and Y axis is the percentage of pairs detected as dissimilar. For Twitter, the percentage of pairs detected as dissimilar is 34% for r=4 compared to 17% for r=1. For ClueWeb, 19% of the total pairs under comparison are detected as

dissimilar with r=3 while 10% for r=1. The results show that choosing r as 3 or 4 is most effective.

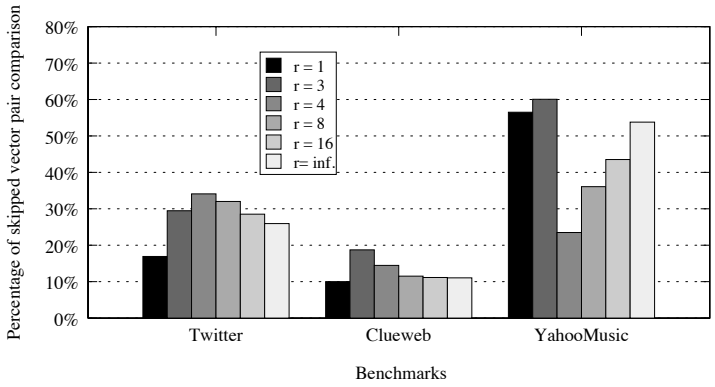


Fig. 17: Percentage of skipped comparison after static partitioning with different r-norms.

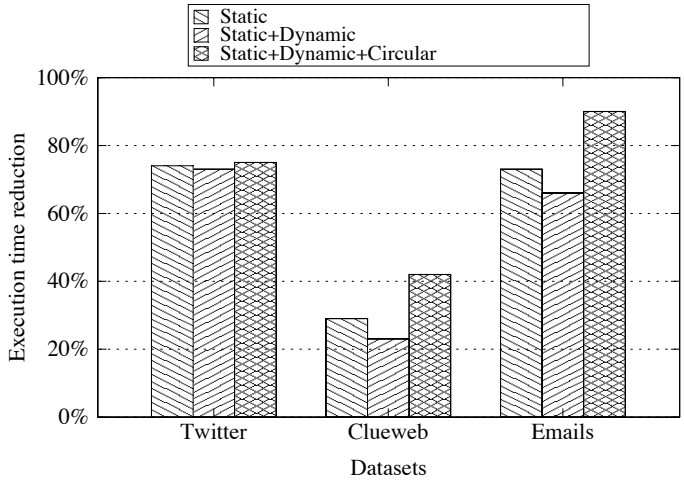


Fig. 18: Reduction percentage of execution time when incorporating different optimization techniques.

Figure 18 shows the percentage of execution time reduction by using different optimization techniques. The number of cores allocated is 120 for the Twitter and Clueweb, and 20 cores for Email. We use static partitioning to detect dissimilarity of vectors, form tasks with the circular load balancing strategy, and adopt dynamic computation filtering when the comparison of two vectors is detected to be unnecessary at runtime [Bayardo et al. 2007; Xiao et al. 2008]. Each bar represents execution time reduction ratio by using one or more optimization techniques compared to the baseline. “Static” implies using static filtering over the baseline algorithm without additional dynamic filtering. “Static+Dynamic” uses static filtering after partitioning and dynamic filtering over the baseline. “Static+dynamic+Circular” means that all three

techniques are applied. Overall reduction from using all techniques leads to 75% for Twitter, 42% for Clueweb, and 90% for Emails. Static partitioning with dissimilarity detection leads to about 74% reduction for Twitter, about 29% for Clueweb, and about 73% for Emails. Adding the circular task mapping contributes an additional 2% for Twitter, 19% for Clueweb, and 14% for Emails. For this case, dynamic computation filtering after static elimination actually slows down the computation in 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. Notice that features vectors of these datasets all use non-binary weights and we find that dynamic filtering integrated in our scheme does improve performance for binary vectors.

10.3. Performance Difference of PSS, PSS1 and PSS2

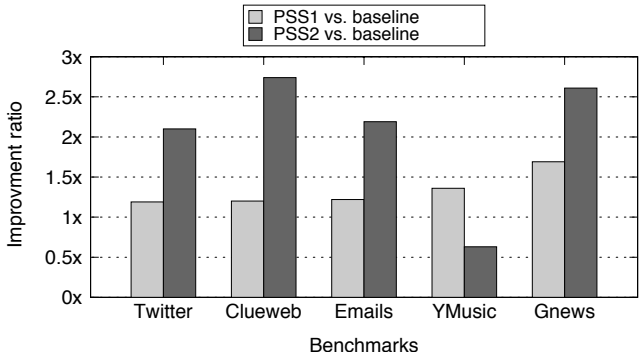


Fig. 19: Y axis is ratio $\frac{Time_{PSS}}{Time_{PSS1}}$ and $\frac{Time_{PSS}}{Time_{PSS2}}$. The average task running time includes I/O.

In this subsection, we compare the performance of PSS1 and PSS2 with the baseline PSS using multiple application benchmarks. We observe the same trend that PSS1 outperforms the baseline, and PSS2 outperforms PSS1 in all cases except for the YMusic benchmark. Figure 19 shows the improvement ratio on the average task time after applying PSS1 or PSS2 over the baseline PSS. Namely $\frac{Time_{PSS}}{Time_{PSS1}}$ and $\frac{Time_{PSS}}{Time_{PSS2}}$. PSS is cache-oblivious and each task handles a very large partition that fits into the main memory (but not fast cache). For example, each partition for Clueweb can have around 500,000 web pages. Result shows PSS2 contributes significant improvement compared to PSS1. For example, under Clueweb dataset, PSS1 is 1.2x faster than the baseline PSS while PSS2 is 2.74x faster than PSS. The split size s for PSS1 and s and b for PSS2 are optimally chosen.

While PSS1 outperforms PSS in most datasets, there is an exception for Yahoo! music benchmark. In this case, PSS1 is better than baseline, which is better than PSS2. This is due to the low sharing pattern in Yahoo! music dataset. The benefits of PSS2 over PSS1 depend on how many features are shared in area B. Figure 20 shows the average and maximum number of features shared among b vectors in area B, respectively. Sharing pattern is highly skewed and the maximum sharing is fairly high. On the other hand, the average sharing value captures better on the benefits of coalescing. The average number shared exceeds 2 or more for all data when b is above 32 (the optimal b value for PSS2) except Yahoo! music. In the Yahoo! music data, each vector represents a song and features are the users rating this song. PSS2 slows down the execution due to the relatively low level of interest intersection among users.

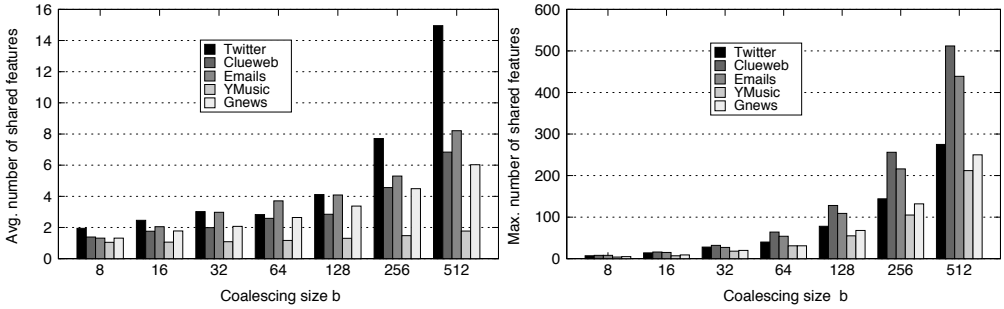


Fig. 20: The left is the average number of shared features among b vectors. The right is the maximum number of features shared among b vectors.

10.4. Cache Behavior and Cost Modeling for PSS1

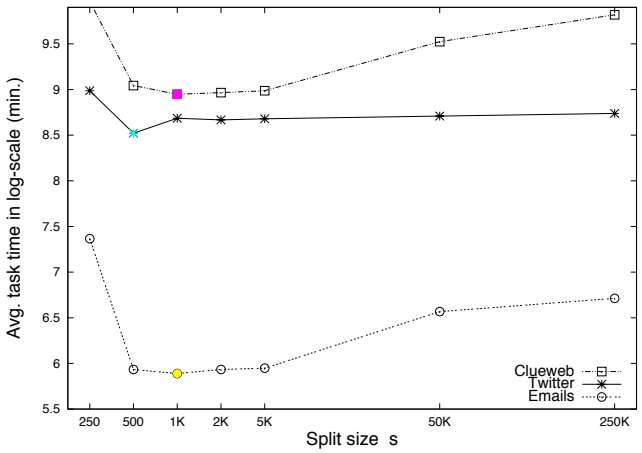
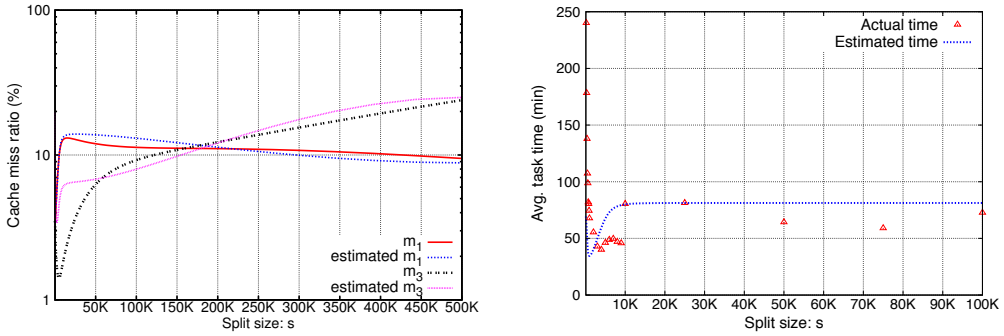


Fig. 21: The average running time in $log\ scale$ per PSS1 task under different values for split size s. The partition size S for each task is fixed, $S = s \times q$.

The gain from PSS to PSS1 is achieved by the splitting of the hosted partition data. Figure 21 shows the average running time of a PSS1 task including I/O in log-scale with different values of s. Notice that the partition size ($S = s \times q$) handled by each task is fixed. The choice of split size s makes an impact on data access cost. Increasing s does not change the total number of basic multiplications and additions needed for comparison, but it does change the traversal pattern of memory hierarchy and thus affects data access cost. For all the datasets shown, the lowest value of the running time is achieved when s value is ranged between 0.5K and 2K, consistent with our analytic results.

We demonstrate the cache behavior of PSS1 modeled in Section 6.2 with the Twitter dataset.

Figure 22(a) depicts the real cache miss ratios for L1 and L3 reported by perf, as well as the estimated L1 miss ratio which is D_1/D_0, and the estimated L3 miss ratio which is D_3/D_2. L1 cache miss ratio grows from 3.5%, peaks when $s = 8K$, and gradually drops to around 9% afterwards when s value increases. L3 cache miss ratio starts from 3.65% when s=100, reaches the bottom at 1.04% when s= 5K, and rises to almost 25%



(a) (b)

Fig. 22: Estimated and real cache miss ratios (a) for PSS1 tasks. Actual vs. estimated average task time (b) for PSS1 in 3M Twitter dataset while split size varies.

when s= 500K. The figure shows that the estimated cache miss ratio approximates the trend of the actual cache miss ratio well.

To validate our cost model, we compare the estimated cost with experimental results in Figure 22(b). Our estimation of cache miss ratios fits the real ratios quite well, reasonably predicts the trend of ratio change as split size changes. When s is very small, the overhead of building and searching the inverted indexes are too high and thus the actual performance is poor. When s ranges from 50K to 80K, the actual running time drops. This is because as s increases, there is some benefit for amortizing the cost of inverted index lookup. Both the estimated and real time results suggest that the optimum s value is around 2K. Given the optimum s, PSS1 is twice faster than when s is 10K.

10.5. Impact of Parameters and Cache Behavior for PSS2

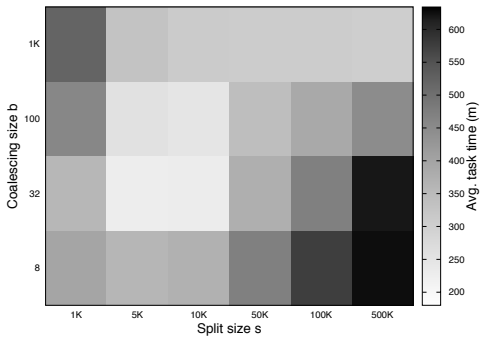


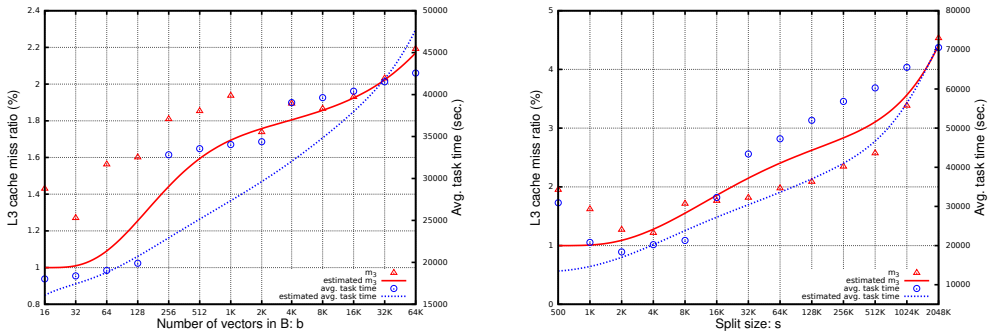
Fig. 23: Each square is an $s \times b$ PSS2 implementation (where $\sum s = S$) shaded by its average task time for Twitter dataset. The lowest time is the lightest shade.

The gain of PSS2 over PSS1 is made by coalescing visits of vectors in B with a control. Figure 23 depicts the average time of the Twitter tasks with different s and b, including I/O. The darker each square is, the longer the execution time is. The shortest

Table VI: Optimal parameters for PSS1 and PSS2 on AMD or Intel architecture.

Architecture	Estimated			Actual		
	PSS1	PSS2		PSS1	PSS2	
	s	s	b	s	s	b
AMD	3,472	2,315	32	4,000	2,000	32
Intel	2,604	1,736	32	4,000	4,000	32

running time is achieved when $b = 32$ and s is between 5K to 10K. When b is too small, the number of features shared among b vectors is too small to amortize the cost of coalescing. When b is too big, the footprint of area C and B becomes too big to fit into L2 cache.



(a) (b)

Fig. 24: Estimated and real L3 cache ratios of PSS2 given s=2K with different b (a) and given b=32 with different s (b). Experiment uses Twitter benchmark with 256K vectors in each partition ($s \cdot q$=256K).

Figure 24 compares the estimated and real L3 cache ratios, as well as average task running time. When s is fixed as 2K records, optimal b is shown as 32 for both cache miss ratio and running time. When b is fixed as 32 records, $s = 2K$ provides the lowest point in cache miss ratio and running time. When s or b are chosen larger than the optimal, running time increases due to higher cache miss ratio. Our analytical model correctly captured the trend and optimal values.

Table VI lists the optimal parameters for PSS1 and PSS2 on AMD and Intel machines we have tested. As an example, here we illustrate how to calculate the optimal parameters for PSS2 on AMD machines. As explained in Section 7, the optimal case is achieved when S_i, B, C all fit in L2 cache, i.e. $S_i + B + C \leq L2\ capacity$.

Similar to the results reported for AMD architecture in the other subsections, we observe 3.7x speedup for PSS1 over cache-oblivious PSS, and 3.6x speedup for PSS2 over PSS1.

Notice such computation could be affected by the workload. For example, when the L2 cache is shared among two cores, and both cores are running cache-intensive computations, L2 cache size in effect is reduced to 1MB. With other parameters fixed, the

optimal case is reduced by half when twice as many share-cache processes are running. Reduced range means the same amount of vectors originally fit in faster cache, now needs to be swapped out and introduces a cache miss.

10.6. Approximated APSS with LSH and PSS

We assess the benefit of integrating PSS with LSH when conducting approximated APSS. Table VII reports the runtime breakdown of conducting APSS for 20M Tweets with 95% target recall for all pairs with cosine similarity over 0.95 using 50 cores. Notice that when a higher value of k is used, the more time is spent on sequential LSH computation, including computing random projection and data copy. When a relatively lower value of k is used, the majority time is spent on the actual similarity comparison conducted in parallel within each bucket. This is because when signature bits k is used in LSH step, each round of input data is split to 2^k buckets after applying k hash functions. When a relatively high value of k is chosen, each data split becomes too small and the cost of data split and data copy contribute to a higher overhead. For the case that applies 4 rounds LSH with 9-bit signature random projection, incorporating LSH method takes 276 minutes in total and computes all pairs similarity with 100% precision and 98.1% recall.

k	l	Time (minute)		
		LSH	PSS	Total
5	3	118	445	563
7	3	135	145	280
9	4	202	74	276
11	4	220	93	313

Table VII: Runtime breakdown of conducting approximated APSS for 20M Tweets with 95% target recall for all pairs with cosine similarity τ over 0.95 using 50 cores.

Table VIII shows the tradeoff of running time with precision and recall when using the integrated LSH/PSS method, pure LSH, and pure PSS. The pure LSH method with a relatively high number of signature bits (k) could provide over 95% recall ratio with more rounds (l) of LSH, but it is hard to make the precision over 94%. On the other hand, the pure PSS method guarantees 100% precision and recall rate, but it takes 8.8x as much time as the integrated LSH/PSS. The result from this table shows that integration of PSS with LSH improves both recall and precision while saving time in exact APSS within each bucket.

Method	k	l	Time (minute)	Precision	Recall
Pure LSH	10	4	219	0.0014%	97.4%
	15	5	351	1.2%	95.5%
	20	7	590	93.6%	95.5%
	25	10	991	93.7%	96.1%
Pure PSS	–	–	$2,435$	100%	100%
LSH + PSS	9	4	276	100%	98.1%

Table VIII: Comparison of three methods for similarity among 20M Tweets. Experiments are conducted using 50 cores. Precision and recall reported are for all pairs with cosine similarity τ over 0.95.

Table IX reports the runtime breakdown of conducting APSS for 40M ClueWeb data with 95% target recall for all pairs with cosine similarity over 0.95 using 300 cores. Same trend as Twitter data is observed with a trade-off between the number of signature bits k and the number of records in each data bucket. For the case that applies 4 rounds LSH with 11-bit signature, the speedup of using LSH method against the parallel time ($79,845$ hours as extrapolated from Figure 14) is $16,138x$ speedup over 300 cores, which means incorporating LSH method is at least $71x$ faster over parallel time with Partition-based method, assuming 75.6% parallel efficiency as shown in Figure 14. Such speedup demonstrates that incorporating LSH with our partition-based similarity search method makes it more accessible to solve the problem of a much larger size. Table X compares Pure LSH, Pure PSS, and LSH+SSH method for 40M ClueWeb dataset using 300 cores. Pure LSH method with relatively high number of signature bits (k) could provide higher than $> 5\%$ recall with more rounds (l) of LSH, but precision is hard to improve over 94%, and more rounds means longer process time. On the other hand, Pure PSS method guarantees 100% precision and recall rate, but takes $71x$ as much time as our adopted method which applies LSH before PSS.

k	l	Time (minute)		
		LSH	PSS	Total
9	4	108	365	473
11	4	114	182	297
13	5	156	171	327

Table IX: Runtime breakdown of conducting APSS for 40M ClueWeb data with 95% target recall for all pairs with cosine similarity τ over 0.95 using 300 cores.

Method	k	l	Time (minute)	Precision	Recall
Pure LSH	15	5	173	0.13%	95.5%
	20	7	269	92.1%	95.5%
	25	10	446	93.1%	96.1%
Pure PSS	–	–	21,123	100%	100%
LSH + PSS	11	4	297	100%	96.5%

Table X: Comparison of three methods for similarity among 40M ClueWeb dataset. Experiments are conducted using 300 cores. Precision and recall reported are for all pairs with cosine similarity τ over 0.95. Due to resource limitation, estimated running time is marked in gray.

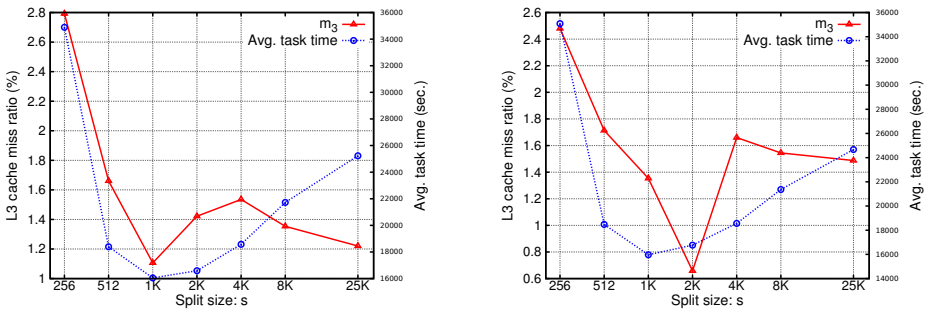
The algorithm implemented in Ivory [Ture et al. 2011] package applies sliding window mechanism on sorted signatures in order to reduce search space, but introduces approximation errors and can at most achieve 0.59 precision and 0.76 recall with $1,000$-bit signatures, 0.74 precision and 0.81 recall with $2,000$-bit signatures, 0.86 precision and 0.78 recall with $3,000$-bit signatures for Jaccard similarity $\tau = 0.3$ [Ture et al. 2011]. With consideration of target precision rate, target recall rate, and the similarity level, we provide a guideline for method choices that meet different requirement and runs relatively fast. We summarize the cases in Table XI. When pairs with very little similarity need to be compared (for example, cosine similarity $\tau < 40\%$), LSH method is not very helpful especially when target recall is high, because the hashing to buckets separates pairs that have low similarity. Depending on the target precision level,

one picks Ivory for lower precision but higher speed, or PSS for higher precision but lower speed. On the other hand, if target recall rate is low, LSH+PSS method is still faster than Ivory or PSS, making it a good choice. For the cases where a modest to high level of similarity level is required, LSH+SSH method is the top choice due to the fast speed, 100% precision, and much higher recall rate it guarantees.

τ	Targeted recall	Target precision	Method choice
Low	Low	-	LSH + PSS
Low	High	Low to modest	Ivory
Low	High	High	PSS
Modest to high	-	-	LSH + PSS

Table XI: A guideline for method choices that meet different requirements of the targeted recall and precision ratios for similarity threshold τ.

10.7. Similarity Measures



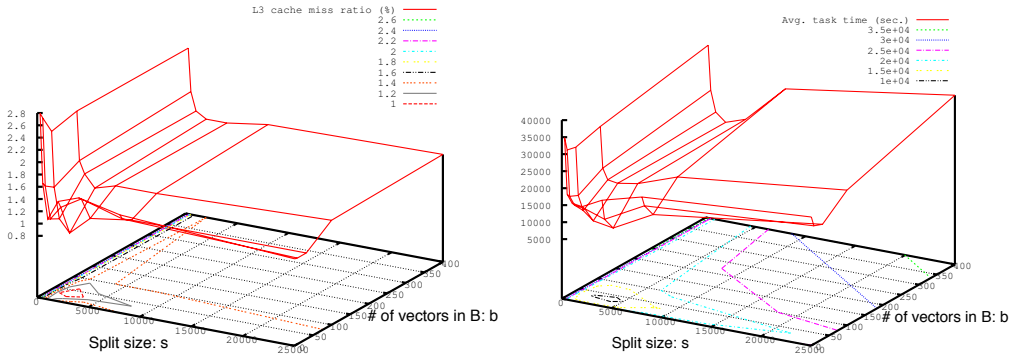
(a) Jaccard coefficient

(b) Dice coefficient

Fig. 25: L3 cache miss ratio m_3 and average task time of PSS1 with different similarity measures. Experiments run on Twitter benchmark with 200K vectors in each partition ($s \times q = 200K$).

We assess the modified PSS1 and PSS2 in handling Jaccard and Dice metrics. Figure 25 shows how the average running time and level 3 cache miss ratios change when different similarity measures are applied. Like the cosine metric, there is a valley in the performance curve and an optimum split point can be chosen. The average task time for Jaccard and Dice metrics are shorter than that for the cosine metric. This is because they use the binary feature values while the cosine metric supports floating-point values. For the three binary similarity measures, the float multiplication is not needed and the value of ψ is smaller. Notice the L3 cache miss ratios are not affected here since ψ is the cost of addition and multiplication.

Figure 26 displays the contour graphs for L3 cache miss ratio m_3 and average task time of PSS2 with Jaccard coefficient measure. Similar to cosine coefficient, Jaccard coefficient algorithm reaches the shortest running time when s is around 4000 and b is around 32. The running time is up-to 3x longer when either s or b is too big or too



(a) (b)

Fig. 26: L3 cache miss ratio m_3 (a) and average task time (b) of PSS2 with Jaccard coefficient measure. Split size s and number of vectors in B b are chosen different values. Experiments run on Twitter benchmark with 200K vectors in each partition.

small. Like the case for the cosine metric, we observe a similar trend in the change of Level 3 cache miss ratio. Our cache performance analysis for choosing parameters of PSS1 and PSS2 with the cosine metric could be applied to Jaccard or Dice similarity.

10.8. Incremental Updates

We evaluate the use of incremental computing with PSS. When new vectors arrive, we accumulate them in a new partition and set the threshold size for triggering PSS as the median size of partitions. Once the new partition grows over the threshold size, a one-to-all PSS-based job is started to compare only the new partition with all original partitions and then new vectors are distributed to partitions based on their dissimilarity relationship. Table XII shows the time cost of our approach with two differnt content change rates using 300 cores when $100K$ tweets are added to a set of $20M$ tweets. We also list the result of the naïve approach that contacts PSS from scratch and the result illustrates the cost advantage of incremental update.

Initial size	Update ratio	Naïve method	Our approach
$20M$ records	0.5%	510 minutes	10 minutes
$20M$ records	5%	558 minutes	57 minutes

Table XII: Runtime comparison between naïve method and our approach for similarity comparison of $100K$ Tweets or $1M$ Tweets update to an original set of $20M$ Tweets using 300 cores.

11. CONCLUSIONS

The main contribution of this paper is a framework of a partitioned similarity search algorithm with cache-conscious data layout and traversal. The partition-based approach yields coarse-grain parallelism, which reduces unnecessary communication and simplifies the runtime comparison of vectors and parallel computation implementation. Static dissimilarity detection filters out unwanted comparisons as earlier as possible and avoids unnecessary I/O, which is most effective for applications using relatively high similarity thresholds. The evaluation shows that PSS can be one or two

orders of magnitude faster than the other parallel solutions for exact APSS. When approximation of APSS is allowed, PSS can be used with LSH for accomplishing high recalls.

The partition-based approach simplifies the runtime computation and allows us to focus on the speedup of inter-partition comparison by exploiting memory hierarchy with a cache-conscious data layout and traversal pattern design. Specifically, we were able to predict the optimum data-split size by identifying the data access pattern, modeling the cost function, and estimating the task execution time. The key techniques are to 1) split data traversal in the hosted partition so that the size of temporary vectors accessed can be controlled and fit in the fast cache; 2) coalesce vectors with size-controlled inverted indexing so that the temporal locality of data elements visited can be exploited. Our analysis provides a guidance for optimal parameter setting. The evaluation results show that the optimized code can be upto 2.74x as fast as the original cache-obvious design. Vector coalescing is effective if there is a decent number of features shared among the coalesced vectors.

Given partitions have different sizes, it will be interesting in the future to study partition-specific optimization. For example, some of parameters in Table I such as k and p_s are estimated based on the entire dataset and partition-specific modeling may enhance performance. Currently the execution of PSS tasks is completely independent and as they share some data partitions, opportunities exist in optimizing shared data movement and cache locality.

ACKNOWLEDGMENTS

We thank Paul Weakliem, Kadir Diri, and Fuzzy Rogers for their help with the computer cluster, and Xifeng Yan and the anonymous referees for their insightful comments. This work is supported in part by NSF IIS-1118106 and 1528041 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.

REFERENCES

Fabio Aiolli. 2013. Efficient Top-n Recommendation for Very Large Scale Binary Rated Datasets. In *Proc. of 7th ACM Conf. on Recommender Systems*. 273–280.

Maha Alabduljalil, Xun Tang, and Tao Yang. 2013a. Cache-Conscious Performance Optimization for Similarity Search. In *Proc. of ACM International Conference on Research and Development in Information Retrieval(SIGIR)*.

Maha Alabduljalil, Xun Tang, and Tao Yang. 2013b. Optimizing Parallel Algorithms for All Pairs Similarity Search. In *Proc. of 6th ACM Inter. Conf. on Web Search and Data Mining (WSDM)*.

David C. Anastasiu and George Karypis. 2014. L2AP: Fast cosine similarity search with prefix L-2 norm bounds. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*. 784–795. DOI:http://dx.doi.org/10.1109/ICDE.2014.6816700

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

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

Ranieri Baraglia, Gianmarco De Francisci Morales, and Claudio Lucchese. 2010. Document Similarity Self-Join with MapReduce. In *ICDM*.

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

Peter Boncz, Data Distilleries B V, Stefan Manegold, and Martin L Kersten. 1999. Database Architecture Optimized for the new Bottleneck : Memory Access. In *VLDB*.

Fidel Cacheda, Víctor Carneiro, Diego Fernández, and Vreixo Formoso. 2011. Comparison of collaborative filtering algorithms: Limitations of current techniques and proposals for scalable, high-performance recommender systems. *ACM Trans. Web* (2011).

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

J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* 16, 1 (March 1990), 1–17. DOI:http://dx.doi.org/10.1145/77626.79170

Iain S. Duff, Michael A. Heroux, and Roldan Pozo. 2002. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Trans. Math. Softw.* 28, 2 (June 2002), 239–267. DOI:http://dx.doi.org/10.1145/567806.567810

Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB*.

Hannaneh Hajishirzi, Wen-tau Yih, and Aleksander Kolcz. 2010. Adaptive Near-duplicate Detection via Similarity Learning. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '10)*. ACM, New York, NY, USA, 419–426. DOI:http://dx.doi.org/10.1145/1835449.1835520

Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC '98)*. 604–613.

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

David Kanter. 2010. MD's Bulldozer Microarchitecture. *realworldtech.com* (2010).

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

David Levinthal. 2009. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. *Intel* (2009).

Jimmy Lin. 2009. Brute force and indexed approaches to pairwise document similarity comparisons with MapReduce. In *SIGIR*.

Stefan Manegold, Peter Boncz, and Martin L. Kersten. 2002. Generic database cost models for hierarchical memory systems. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*.

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

Ahmed Metwally and Christos Faloutsos. 2012. V-SMART-Join : A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors. In *VLDB*, Vol. 5. 704–715.

Gianmarco De Francisci Morales, Claudio Lucchese, and Ranieri Baraglia. 2010. Scaling Out All Pairs Similarity Search with MapReduce. In *8th Workshop on LargeScale Distributed Systems for Information Retrieval (2010)*.

Mehran Sahami and Timothy D. Heilman. 2006. 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)*. 377–386.

Venu Satuluri and Srinivasan Parthasarathy. 2012. Bayesian Locality Sensitive Hashing for Fast Similarity Search. *Proc. VLDB Endow.* 5, 5 (Jan. 2012), 430–441. DOI:http://dx.doi.org/10.14778/2140436.2140440

Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. 1994. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc, 510–521.

Kai Shen, Tao Yang, and Xiangmin Jiao. 2000. S+: Efficient 2D Sparse LU Factorization on Parallel Machines. *SIAM J. Matrix Anal. Appl.* 22, 1 (April 2000), 282–305. DOI:http://dx.doi.org/10.1137/S0895479898337385

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

Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. 2013. Streaming Similarity Search over One Billion Tweets Using Parallel Locality-sensitive Hashing. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1930–1941. DOI:http://dx.doi.org/10.14778/2556549.2556574

Xun Tang, Maha Alabduljalil, Xin Jin, and Tao Yang. 2014. Load balancing for partition-based similarity search. *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval - SIGIR '14* (2014), 193–202. DOI:http://dx.doi.org/10.1145/2600428.2609624

Martin Theobald. 2008. SpotSigs : Robust and Efficient Near Duplicate Detection in Large Web Collections. In *SIGIR*. 1–8.

Ferhan Ture, Tamer Elsayed, and Jimmy Lin. 2011. No free lunch: brute force vs. locality-sensitive hashing for cross-lingual pairwise similarity. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information (SIGIR '11)*. 943–952.

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

Richard Vuduc, James W. Demmel, Katherine A. Yelick, Shoaib Kamil, Rajesh Nishtala, and Benjamin Lee. 2002. Performance optimizations and bounds for sparse matrix-vector multiply. In *ACM/IEEE Conf. on Supercomputing*.

Ye Wang, Ahmed Metwally, and Srinivasan Parthasarathy. 2013. Scalable all-pairs similarity search in metric spaces. *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '13* (2013), 829. DOI:http://dx.doi.org/10.1145/2487575.2487625

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

Shanzhong Zhu, Alexandra Potapova, Maha Alabduljalil, Xin Liu, and Tao Yang. 2012. Clustering and Load Balancing Optimization for Redundant Content Removal. In *Proc. of WWW 2012 (21th international conference on World Wide Web), Industry Track*.