UNIVERSITY OF CALIFORNIA
Santa Barbara

Efficient Similarity Search with Cache-Conscious Data Traversal

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Xun Tang

Committee in Charge:

Professor Tao Yang, Chair

Professor Divyakant Agrawal

Professor Xifeng Yan

March 2015

The Dissertation of
Xun Tang is approved:

Professor Divyakant Agrawal

Professor Xifeng Yan

Professor Tao Yang, Committee Chairperson

March 2015

Efficient Similarity Search

with Cache-Conscious Data Traversal

by

Xun Tang

God, grant me the serenity to accept the

things I cannot change,

The courage to change the things I can,

And the wisdom to know the difference.

– American theologian Reinhold Niebuhr

(1892 – 1971)

Acknowledgements

I owe my deepest gratitude to my Ph.D. adviser, Prof. Tao Yang, for his invaluable advice on selecting research topics, tackling problems under extreme circumstances, and refining solutions and presentation. Without his insight and guidance, the work in this dissertation would not have been possible. I also want to thank my dissertation committee, Prof. Divyakant Agrawal and Prof. Xifeng Yan, for their pertinent and helpful instructions on my research work.

I am grateful to members in my research group, Maha Alabduljali, Aleksandra Potapova, Xin Jin, Wei Zhang and Michael Agun, for providing a stimulating and fun environment in which I learned and grew tremendously. Specifically, I would like to express my gratitude to Maha Alabduljali, with whom I had most technical discussions and research collaboration, for her patience and hard work that make the work more enjoyable. Special thanks to my roommate Yun Teng, for all her trust, help, and advice when I need the most.

Most importantly, my sincere appreciation to my mum Yanqiong Lin and step-father Xingmin Xiong, who live on the other side of the planet, and my father Xiaokang Tang, who lived in the other world. I thank them for raising me, supporting me, encouraging me, and loving me forever.

This dissertation study was supported in part by NSF IIS-1118106. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Equipment access for experiments conducted in this dissertation was supported in part by the Center for Scientific Computing at the CNSI and MRL: an NSF MRSEC (DMR-1121053) and NSF CNS-0960316.

To all of them I dedicate this dissertation.

Curriculum Vitæ

Xun Tang

Education

2014

Master of Science in Computer Science, University of California, Santa Barbara.

2009

Bachelor of Science in Computer Science, Fudan University, Shanghai.

Publications

Xun Tang, Maha Alabduljali, Xin Jin, and Tao Yang, Load Balancing for Partitioned Similarity Search, *in Proceedings of the 35th international ACM SIGIR conference on Research and development in Information* (SIGIR 2014).

Xun Tang, Xin Jin, and Tao Yang, Cache-Conscious Runtime Optimization for Ranking Ensembles, *in Proceedings of the 35th international ACM SIGIR conference on Research and development in Information* (SIGIR 2014).

Maha Alabduljali, **Xun Tang**, and Tao Yang, Cache-Conscious Performance Optimization for Similarity Search, *in Proceedings of the 34th international ACM SIGIR conference on Research and development in Information* (SIGIR 2013).

Maha Alabduljali, **Xun Tang**, and Tao Yang, Optimizing Parallel Algorithms for All Pairs Similarity Search, *in Proceedings of the 6th ACM International Conference on Web Search and Data Mining* (WSDM 2013).

Xun Tang, Maha Alabduljali, and Tao Yang, Partitioned Similarity Search with Cache-Conscious Data Layout and Traversal, to be submitted for publication.

Abstract

Efficient Similarity Search with Cache-Conscious Data Traversal

Xun Tang

Similarity search is important for many data-intensive applications to identify a set of similar objects. Examples of such applications include near-duplicate detection and clustering, collaborative filtering for similarity-based recommendations, search query suggestion, and data cleaning. Conducting similarity search is a time-consuming process, especially when a massive amount of data is involved, and when all pairs are compared. Previous work has used comparison filtering, inverted indexing, and parallel accumulation of partial intermediate results to expedite its execution. However, shuffling intermediate results can incur significant communication overhead as data scales up.

We have developed a fast two-stage partition-based approach for all-pairs similarity search which incorporates static partitioning, optimized load balancing, and cache-conscious data traversal. Static partitioning places dissimilar documents into different groups to eliminate unnecessary comparison between their content. To overcome the challenges introduced by skewed distribution of data partition sizes and irregular dissimilarity relationship in large datasets, we con-

duct computation load balancing for partitioned similarity search, with competitiveness analysis. These techniques can improve performance by one to two orders of magnitude with less unnecessary I/O and data communication and better load balance. We also discuss how to further accelerate similarity search by incorporating incremental computing and approximation methods such as Locality Sensitive Hashing.

Because of data sparsity and irregularity, accessing feature vectors in memory for runtime comparison incurs significant overhead in modern memory hierarchy. We have designed and implemented cache-conscious algorithms to improve runtime efficiency in similarity search. The idea of optimizing data layout and traversal patterns is also applied to the search result ranking problem in runtime with multi-tree ensemble models.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

All Pairs Similarity Search (APSS) [14], which identifies similar objects in a dataset, is used in many applications including collaborative filtering based on user interests or item similarity [3], search query suggestions [46], web mirrors and plagiarism recognition [49], coalition detection for advertisement frauds [43], spam detection [22, 37], clustering [12], and near duplicates detection [33].

The complexity of naïve APSS can be quadratic to the dataset size. In big data computing applications such as web mining with hundreds of millions of objects, improvement in efficiency can have a significant impact to speed up discovery and offer more rich options under the same computing constraint. Previous researches on expediting similarity computing developed filtering [14, 55, 6] and inverted indexing [41, 44] methods. 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 [23] to compute and collect similarity results in parallel using an inverted index [41, 44, 13]. Unfortunately, the cost of communicating the intermediate partial results is still excessive and such solutions are not scalable for larger datasets.

We pursue a design [5] that conducts partition-based APSS 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 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 the computation when calculating the similarity of vectors from two partitions, we further consider the impact of memory hierarchy on the execution time. The main memory access latency can be 10 to 100 times slower than the L1 cache latency. Thus, 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 fast 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, such that core data can fully reside in the fast cache. This approach outperforms other approaches [41, 13] by an order of magnitude due to the simplified parallelism management and aggressive elimination of unnecessary I/O or comparison.

All-pairs Similarity Comparisons can be performed through a number of independent tasks where each compares a partition of vectors with other candidate vectors [5]. Given a large number of data partitions, we need to assign them to parallel machines and decide the direction of similarity comparison due to the symmetric property of comparison. It is challenging to balance the computation load among parallel machines with a distributed architecture. Load imbalance can hugely affect scalability and overall performance. This is mainly due to the variation in partition sizes and irregular dissimilarity relationship in large datasets.

We developed a two-stage assignment algorithm that reduces the network load and balances the similarity computation across the parallel tasks. The first stage constructs a preliminary load assignment over tasks. The second stage refines the

assignment to be more balanced. Since comparison tasks typically spend much more time in computation than in I/O and communication, our analysis shows that the developed solution is competitive to the optimum with a constant ratio. We further improve the dissimilarity detection ability in the static partitioning step, while producing partitions with relatively even sizes to facilitate the load balancing step. The evaluation results show that the proposed scheme outperforms a previously developed solution by up to 41% in the tested cases.

Once a set of distinguished result pages have been selected after conducting All-pairs Similarity Search on all the web pages that match the query, a ranking among the result pages needs to be computed before the final search result page could be presented to the users. To organize the search result page in a way that maximizes the total reward, instead of relying on human judges, many companies adopt a system that leverages implicit user feedback to build machine-learned models to generate ranking score for each record. Tree-based learning ensembles are trained offline and applied online to serve hundreds of millions of queries live each day [36].

Learning ensembles based on multiple trees are effective for web search and other complex data applications (e.g. [27, 20, 28]). It is not unusual that algorithm designers use thousands of trees to reach better accuracy and the number of trees becomes even larger with the integration of bagging. For example, winning teams

in the Yahoo! learning-to-rank challenge [20] have all used boosted regression trees in one form or another and the total number of trees reported for scoring ranges from 3,000 to 20,000 [30, 18, 32], or even reaches 300,000 or more combined with bagging [45].

Generally speaking, application training data with less attributes may require smaller trees or a smaller number of trees. But as complex applications evolve over the time, more attributes are augmented and using more trees usually yields better accuracy. Using a large number of trees can improve accuracy, but it takes time to calculate ranking scores of matched documents. We investigate data traversal methods for fast score calculation with a large ensemble when ranking a modest number of matched documents.

We propose a 2D blocking scheme for better cache utilization with simpler code structure compared to previous work. Our experiments show that 2D blocking can be up to 620% faster than DOT, up to 245% faster than SOT, and up to 50% faster than VPred. After applying 2D blocking on top of VPred which shows advantage in reducing branch mis-prediction, the combined solution Block-VPred could be up to 100% faster than VPred. The experiments with several benchmarks show significant acceleration in score calculation without loss of ranking accuracy.

This thesis is organized as follows. This chapter and Chapter 2 serve as the introduction and background of the three major components of this thesis. The

following three chapters contribute to each of them. In Chapter 3, we present the design and implementation of cache-conscious algorithms for Partitioned All-pairs Similarity Search. We also discuss in this chapter how to further accelerate similarity search by incorporating incremental computing and approximation methods such as Locality Sensitive Hashing. In Chapter 4, we discuss our technique to mitigate the load balance problem in similarity search, and how efficient our technique is comparative to the optimum. In Chapter 5, we present a technique for efficient search result ranking in the runtime system, with tree-based learning ensembles. We conclude this thesis and discuss the future work in Chapter 6.

Chapter 2

Background and Related Work

Our approach implements a fast two-stage partition-based algorithm for all-pairs similarity search, and incorporates the techniques of static partitioning, optimized load balancing, and cache-conscious data traversal. In this chapter, we first explore the related work in each sub-domain. After the similarity search and de-duplication, the matched documents are ranked in relevance and preference before presented to user. In this chapter, we also cover the background and related work of runtime search result ranking, and how our approach differs from the others.

Following the work in [14], 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 vectors is computed as:

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

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 APSS is high for a big dataset. There are application-specific methods applied to reduce the complexity. For example, text mining removes stop-words or features with extremely high document vector frequency [12, 26, 41]. We adopt these methods in the pre-processing step throughout our experiments.

There are several groups of optimization techniques developed in the 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 without complete derivation of final similarity scores [14, 55, 44].

Similarity-based grouping in data pre-processing. The search scope for similarity can be reduced when potentially similar vectors are placed in one group. One can use an inverted index [55, 41, 44] developed for information retrieval [12]. This approach identifies vectors that share at least one feature as potentially similar, so certain data traversal is avoided. Similarly, the work in [52] 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) can be considered as grouping similar vectors into one bucket with approximation [31, 51]. This approach has a trade-off between precision and recall, and may introduce redundant computation when multiple hash functions are used. A study [6] shows that exact comparison algorithms can deliver performance competitive to LSH when computation filtering is used. In partition-based APSS [5], dissimilar vectors are identified in the static partitioning step. The APSS problem is then converted to executing a set of independent tasks each compares one partition with some of the other partitions. These tasks can be executed in parallel with much simplified parallelism management.

Load balancing and scheduling. Exploiting parallel resources over thousands of machines for scalable performance is important and challenging. Load balancing is considered in the context of search systems for index serving [11, 39]. A recent study [54] introduces a division scheme to improve load balance for dense APSS problems using multiple rounds of MapReduce computation. In order to minimize the communication overhead while maintaining the computational load balance, this thesis focuses on load balancing of APSS with record-based partitioning. The general load balancing and scheduling techniques for clusters and parallel systems have been extensively addressed in previous work. A simple greedy policy [29] that maps a ready task to a computation unit once it becomes

idle is widely adopted (e.g. [15]). Scheduling for MapReduce systems such as Hadoop [23, 56] has followed the greedy policy to execute queued tasks on available cores and exploit data locality whenever feasible. Assuming that parallel tasks are scheduled following such a greedy policy, we address how these tasks should be formed considering scalability and efficiency.

Cache-conscious data traversal. Cache optimization for computationally intensive applications is studied in the context of general database query processing [47, 16]. In particular, the problem of hash join in a main memory DBMS has attracted much attention. Radix-cluster [42] 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, while the computation studied in this thesis focuses on similarity search involving many common features among vector pairs. Cache optimization for computationally intensive applications is studied in the context of matrix-based scientific computing [25, 24, 53, 48]. Motivated by these studies, we investigate the opportunities of cache-conscious optimization targeting APSS problem.

Search result ranking in the runtime system. Computing scores from a large number of trees is time-consuming. Access of irregular document attributes along with dynamic tree branching impairs the effectiveness of CPU cache and

instruction branch prediction. Compiler optimization [10] cannot handle complex code such as rank scoring very well. For example, processing a 8,051-tree ensemble can take up to 3.04 milliseconds for a document with 519 features on an AMD 3.1 GHz core. Thus the scoring time per query exceeds 6 seconds to rank the top-2,000 results. It takes more time proportionally to score more documents with larger trees or more trees, and this is too slow for interactive query performance. Multi-tree calculation can be parallelized; however, query processing throughput is not increased because less queries are handled in parallel. Trade-off between ranking accuracy and performance can be played by using earlier exit based on document-ordered traversal (DOT) or scorer-ordered traversal (SOT) [19], and by tree trimming [7]. The work in [8] proposes an architecture-conscious solution called VPred that converts control dependence of code to data dependence and employs loop unrolling with vectorization to reduce instruction branch mis-prediction and mask slow memory access latency. The weakness is that cache capacity is not fully exploited and maintaining the lengthy unrolled code is not convenient. Unorchestrated slow memory access incurs significant costs since memory access latency can be up to 200 times slower than L1 cache latency. How can fast multi-tree ensemble ranking with simple code structure be accomplished via memory hierarchy optimization, without compromising ranking accuracy? We propose a cache-conscious 2D blocking method to optimize data traversal for better temporal

cache locality. The proposed techniques are complementary to previous work and can be integrated with the tree trimming and early-exit approximation methods.

Chapter 3

Cache-Conscious Partition-based Similarity Search

3.1 Partition-based Similarity Search Framework

The framework for Partition-based Similarity Search (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. These tasks are independent when running on a set of parallel machines. Figure 3.1 depicts the whole process.

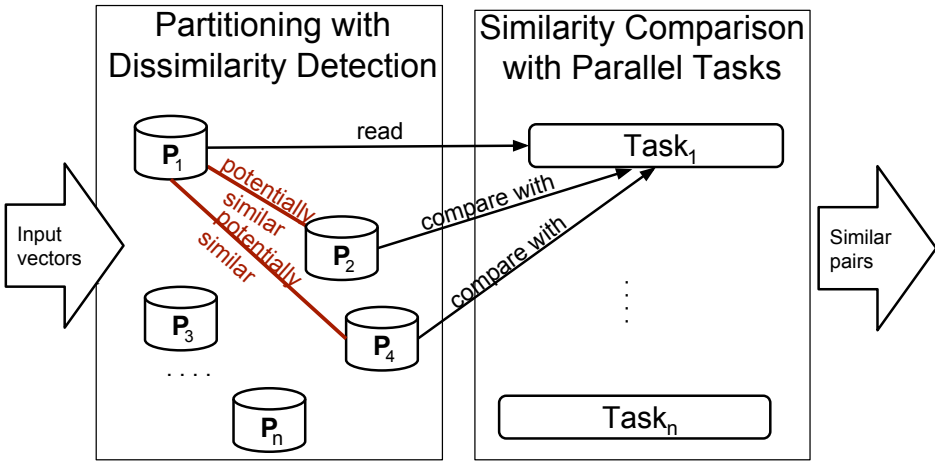


Figure 3.1: Illustration of partition-based similarity search.

Algorithm 1 Definition of each PSS task T_k.

Read all vectors from assigned partition P_k.

Build inverted index of these vectors.

Conduct self-comparison among vectors in P_k.

repeat

Fetch a potentially similar partition.

for $d_j \in$ fetched partition **do**

COMPARE (P_k, d_j).

end for

until all non-dissimilar partitions are fetched.

Dissimilarity-based partitioning identifies dissimilar vectors without explicitly computing the product of their features. One approach [5] utilizes the following inequality that calculates the 1-norm and ∞-norm of each vector:

$$Sim(d_i, d_j) \leq min(||d_i||_\infty ||d_j||_1, ||d_j||_\infty ||d_i||_1) < \tau.$$

The partitioning algorithm sorts the vectors based on their 1-norm values first. It then uses the sorted list to identify dissimilar pairs (d_i, d_j) satisfying inequality $\|d_i\|_1 < \frac{\tau}{||d_j||_\infty}$. A different τ value would affect the outcome of the dissimilarity-based partitioning.

Once the dataset is separated into v partitions, v independent tasks are scheduled. 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 the memory of one machine as the data partitioning can be adjusted to satisfy such condition. Other partitions to be compared with may not fit the remaining memory and need to be fetched gradually from a local or remote storage. In a computing cluster with a distributed file system such as Hadoop, tasks can seamlessly fetch data without concerning about the physical locations of data.

Algorithm 1 describes the function of each task T_k in partition-based similarity search. Task T_k loads the assigned partition P_k and produces an inverted index to be used during the partition-wise comparison. Next, T_k fetches a number of vectors from potentially similar partitions and compares them with the local

partition P_k one by one. Fetch and comparison is repeated until all candidate partitions are processed.

3.2 Runtime Data Layout

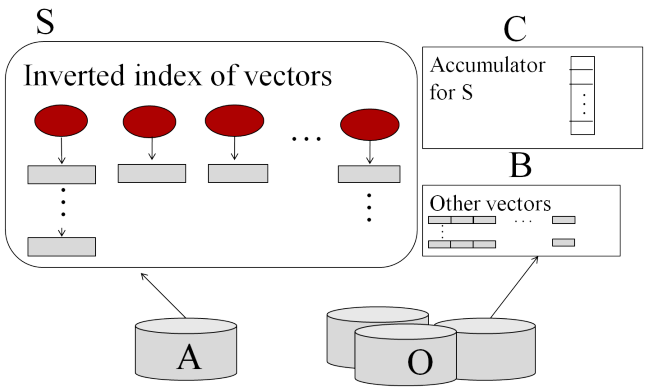


Figure 3.2: A PSS task compares the assigned partition A with other partitions O.

Figure 3.2 depicts a task for partition-based similarity search interacting with a CPU core with multiple levels of cache. Two or three cache levels are typical in today's Intel or AMD architecture [40, 38]. We assume that the assigned partition A fits in the memory of one machine as the data partitioning can be adjusted to satisfy such an assumption. But vectors in O can exceed memory and need to be fetched gradually from a local or remote storage. In a computer cluster with

distributed file system such as Hadoop, a task 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 3.2. 1) Area S: hosts the assigned partition A. 2) Area B: stores a block of vectors fetched from other candidate partitions O at each comparison step. 3) Area C: stores intermediate results temporarily.

Algorithm 2 PssTask(A, O)

1: Input: Partition A assigned to the task, and other candidate partitions O.
2: Output: Similar pairs and their corresponding similarity score.
3: Read all vectors from assigned partition A into S.
4: Build inverted index of these vectors and store in S.
5: **repeat**
6: Fetch a set of vectors from O into B.
7: **for** $d_j \in B$ **do**
8: PssCompare(S, d_j).
9: **end for**
10: **until** All vectors in O are fetched.

Algorithm 2 and Function 3 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

Algorithm 3 PssCompare(S, d_j)

Initialize array *score* of size |S| with zeros.
$r_j = ||d_j||_1$.
for $t \in d_j$ and $posting(t) \in S$ **do**
    for $d_i \in posting(t)$ and d_i is a candidate **do**
        $score[i]$=$score[i]$+$w_{i,t}\times w_{j,t}$.
        if $(score[i]+||d_i||_\infty \times r_j < \tau)$ **then**
            Mark d_i as non-candidate.
    end for
    $r_j = r_j - w_{j,t}$.
end for
for $i = 1$ **to** |S| **do**
    if $score[i] \geq \tau$ **then**
        write $(d_i, d_j, score[i])$.
end for

weights, stored in a compact manner. After loading the assigned vectors, the task inverts them locally and stored 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 7 in Algorithm 2). 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 3). 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 12 in Function 3) 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 Function 3) 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 negative value of $score[i]$ indicates d_i marked as an non-candidate.

3.3 PSS1: 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 there is a potential problem that the accessed areas S or C may not fit in the fast cache. In that case, temporal locality is not exploited, meaning the second

access of the same element during any computation will be a cache miss. As shown in the next section, this leads to frequent slow memory access and a significant increase in execution time. Since fast access of each area S, B or C is equally important in the core computation (Lines 5 and 6 in Function 3), one idea is to let area C fit in fast cache by explicitly dividing vectors of the assigned partition in S into a set of splits and have the task focus on one split at a time.

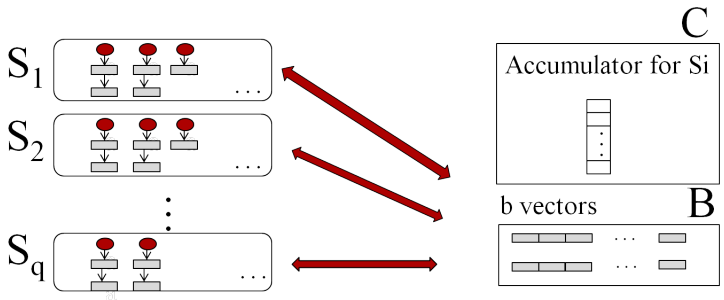


Figure 3.3: A partition in area S is further divided into multiple splits for each PSS1 task.

Figure 3.3 and 3.4 illustrate this cache-conscious data splitting idea. The corresponding algorithm called PSS1 is shown in Algorithm 4. First, it divides the hosted vectors in S into q splits. 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 b_j in B. The access in area C is localized such that array $score[\ ]$ and $||d||_\infty[\ ]$ can fully fit in L1 cache. This improves temporal locality of data elements for area

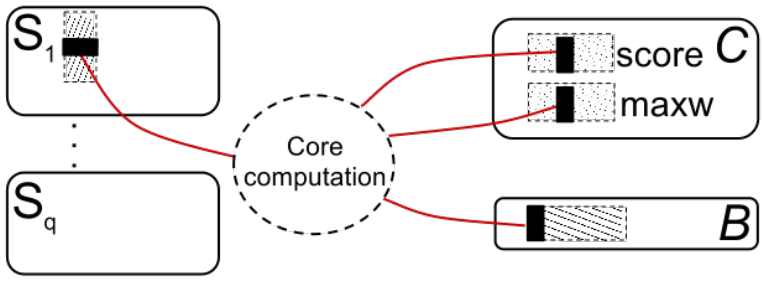


Figure 3.4: Core computation in PSS1 and its interaction with data items. Four data items are involved in the core computation. The striped area indicates cache coverage.

Algorithm 4 Pss1Task(A, O)

1: Input: Partition A assigned to the task, and other candidate partitions O.
2: Output: Similar pairs and their corresponding similarity score.
3: Read and divide A into q splits.
4: Build an inverted index for each split and store in area S_i.
5: **repeat**
6: Fetch a set of vectors from O into B.
7: **for** $d_j \in B$ **do**
8: **for** $S_i \in S$ **do**
9: PssCompare(S_i, d_j).
10: **end for**
11: **end for**
12: **until** All vectors in O are fetched.

C and reduces the access time by an order of magnitude. The core computation speeds up as a result.

The data splitting also introduces potential benefits from exploiting the multi-core CPU architecture via threads. Every time a data block from O is fetched into B, there can be multiple threads running in parallel to execute function COMPARE(S_i, d_j) (Line 9 in Algorithm 4) where d_j is a vector in B.

The question is, how to determine the s value of each split so that the caches are best utilized? This is discussed next.

3.4 PSS1: Cache Performance and Cost Analysis

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 3.1 describes the parameters used in our analysis. They represent the characteristics of the given dataset, algorithm variables, and the system setting.

3.4.1 Task Execution Time

The total execution time for each task contains two parts: I/O and computation. I/O cost occurs for loading the assigned vectors A, fetching other potentially

Table 3.1: 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 look-up 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	Pre-fetch 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

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 here.

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 3.1, h is the cost of looking up the posting of a feature appeared in a vector in B. p_s denotes the average length of postings visited in S_i (only when a common feature exists), and p_s estimates the number of iterations for Line 3 in Function 3. 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 4) can be approximately modeled

as follows.

$$\text{Time} = q \left[nk(\overbrace{h}^{look-up} + \overbrace{p_s \times 2\psi}^{multiply+add}) + \overbrace{\delta_{total}}^{traverse\ S,B,C} \right]. \tag{3.1}$$

As s increases, q decreases and the cost of inverted index look-up 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.

3.4.2 Memory and Cache Accesses of PSS1

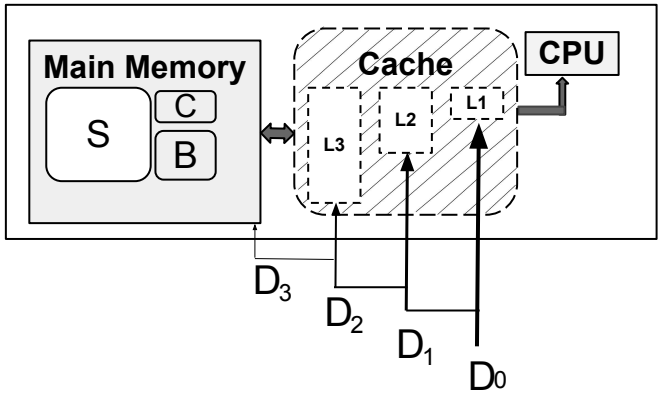


Figure 3.5: Data access misses for three-layer cache hierarchy, where $D_{j-1} \geq D_j$, j=1, 2, 3.

Here we estimate the cost of accessing data in S_i, B, and C. As illustrated in Figure 3.5, D_0 is defined as the total number of data accesses in performing COMPARE(S_i, d_j) in Algorithm 4. 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.

$$\delta_{total} = (D_0 - D_1)\delta_1 + (D_1 - D_2)\delta_2 + (D_2 - D_3)\delta_3 + D_3\delta_{mem}. \tag{3.2}$$

To conduct the computation in Lines 5 and 6 of Function 3, 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:

$$D_0 = D_0(S_i) + D_0(B) + D_0(C) = \overbrace{nkp_s}^{S_i} + \overbrace{nkp_s}^{B} + \overbrace{2nkp_s}^{C}. \tag{3.3}$$

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.

$$\begin{aligned} D_j =& D_j(S_i) + D_j(B) + D_j(C) \\ =& D_{j-1}(S_i) * m_j(S_i) + D_{j-1}(B) * m_j(B) + D_{j-1}(C) * m_j(C). \end{aligned} \tag{3.4}$$

Table 3.2 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

Table 3.2: 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 $m_j(\mathcal{C})$ for accessing data in $\mathcal{C}$.

Case	$\mathbf{m_1}$		$\mathbf{m_2}$		$\mathbf{m_3}$		**Description**
	$\mathbf{S_i}$	**C**	$\mathbf{S_i}$	**C**	$\mathbf{S_i}$	**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.

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 the 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 CPU pre-fetches a few cache lines in advance, in anticipation of using consecutive memory regions [40, 38]. 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 pre-fetch factor f. Let f be the effective pre-fetch 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 3.2.

- 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 we have

$$\begin{aligned}\delta_{total} &= (D_0 - D_1)\delta_1 + D_1\delta_2 \\ &= \left[\ nkp_s(1 - \max(\frac{1}{p_s}, \frac{e_s}{fl})) + nkp_s + 2nkp_s(1 - \frac{e_c}{fl})\ \right]\delta_1 \\ &+ \left[\ nkp_s \max(\frac{1}{p_s}, \frac{e_s}{fl}) + 2nkp_s\frac{e_c}{fl}\ \right]\delta_2.\end{aligned} \tag{3.5}$$

  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 3.2. 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. So ideally, data access of PSS1 can be $10x$ 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 3.6 shows the actual data-access-to-computation ratio collected from our experiment using Twitter dataset when s varies from 100 to 25K. We measure the ratio of the data access time (including

the inverted index look-up) 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.

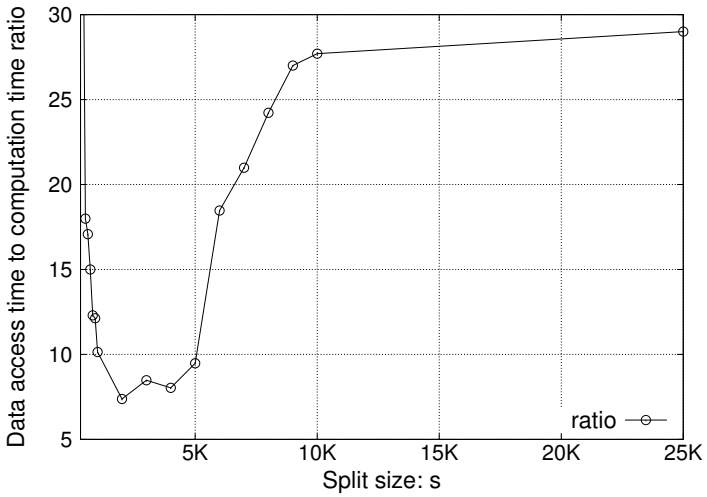


Figure 3.6: Y axis is the ratio of actual data access time to computation time for Twitter data observed in our experiments.

3.5 PSS2: 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 every time. 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.

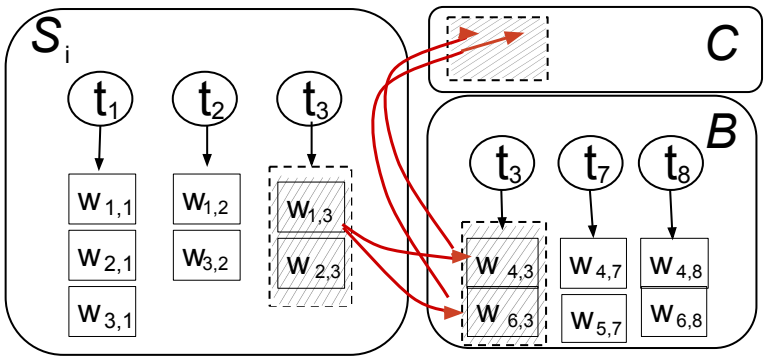


Figure 3.7: Example of data traversal in PSS2. Five data items are involved in the core computation. The striped area indicates coverage of a cacheline.

Figure 3.7 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 5 Pss2Task(A, O).

1: Input: Partition A assigned to the task, and other candidate partitions O.
2: Output: Similar pairs and their corresponding similarity score.
3: Read A and divide it into q splits of s vectors each.
4: Build an inverted index for each split S_i.
5: **repeat**
6: Fetch b vectors from O and build inverted index in B.
7: **for** $S_i \in S$ **do**
8: Pss2Compare(S_i, B).
9: **end for**
10: **until** All vectors in O are compared.

Algorithm 5 and Function 6 describe a PSS2 task. The key distinctions from a PSS1 task are as follows.

Algorithm 6 Pss2Compare(S_i, B).

Initialize array $score$ of size $s \times b$ with zeros.
for $j = 1$ **to** b **do**
    $r[j] = ||d_j||_1$.
end for
for feature t appears in both B and S **do**
    for $d_i \in posting(t)$ in S **do**
        for $d_j \in posting(t)$ in B and d_i is a candidate **do**
            $score[i][j]$=$score[i][j]$+$w_{i,t}$×$w_{j,t}$.
            if ($score[i][j]$+$||d_i||_\infty$×$r[j]$<τ) **then**
                Mark pair d_i and d_j as non-candidate.
        end for
    end for
    for $d_j \in posting(t)$ in B **do**
        $r[j] = r[j] - w_{j,t}$.
    end for
end for

Algorithm 6 Pss2Compare(S_i, B) (continued).

```
1: for i = 1 to s do
2:     for j = 1 to b do
3:         if score[i][j] ≥ τ then
4:             Write (d_i, d_j, score[i][j]).
5:     end for
6: end for
```

- 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 8 in Algorithm 5).

- 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 6).

- The above approach also benefits the amortization of inverted index look-up cost. In PSS1, every term posting look-up for S_i only benefit the 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 the choice of b.

In the next subsection, we will explain in detail why the parameter choice affects the cache utilization, how the parameter choice changes the cache miss ratios by example cases, and generalize the cases in a cache analytic model. For simplicity of presentation, the analysis is applied to PSS2 without considering dynamic elimination (line 6 and line 7 in Function 3).

3.6 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 of 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. If increasing b values expands the size of variables in $\mathcal{B}$ and $\mathcal{C}$ too much, $\mathcal{B}$ 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 3.8) and as a result, a large b value does not linearly increase p_b value.

From cache analysis of PSS1, we expect that PSS2 performs best when S_i, $\mathcal{B}$ and $\mathcal{C}$ fit L2 cache but none of them fit L1 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 in-

equality 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 case 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 3.2.

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

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 cache.

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,

$$\frac{\delta_{total}(PSS1)}{\delta_{total}(PSS2)} = \frac{D_0\delta_1 + D_1(PSS1)(\delta_2 - \delta_1)}{D_0\delta_1 + D_1(PSS2)(\delta_2 - \delta_1)} \lessapprox \frac{D_1(PSS1)}{D_1(PSS2)} = \frac{e_s + 2e_c}{\frac{e_s}{p_b} + \frac{e_b}{p_s} + \frac{3e_c}{p_b}}. \tag{3.6}$$

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 further analyze the cache miss ratio and access time for other cases, and compare their performance in the form of of the ratio of data access time (including the inverted index look-up time) over the computation time $\frac{\text{Data-access}}{\text{Computation}}$.

Figure 3.8 plots the data-access-to-computation ratio ratio for the different cases of parameters in PSS1 and PSS2 cases are from Table 3.3 when handling the Twitter dataset. This figure confirms that PSS2 reaches the lowest ratio when S_i, $\mathcal{B}$ and $\mathcal{C}$ fit L2 cache, and its data access speed can be up-to 14x faster than the others.

Figure 3.8 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

Table 3.3: Explanation of case abbreviations in Figure 3.8.

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.

small. It demonstrates the advantage of PSS2 over PSS1 (a significant reduction of the task execution time) by exhibiting good reference locality.

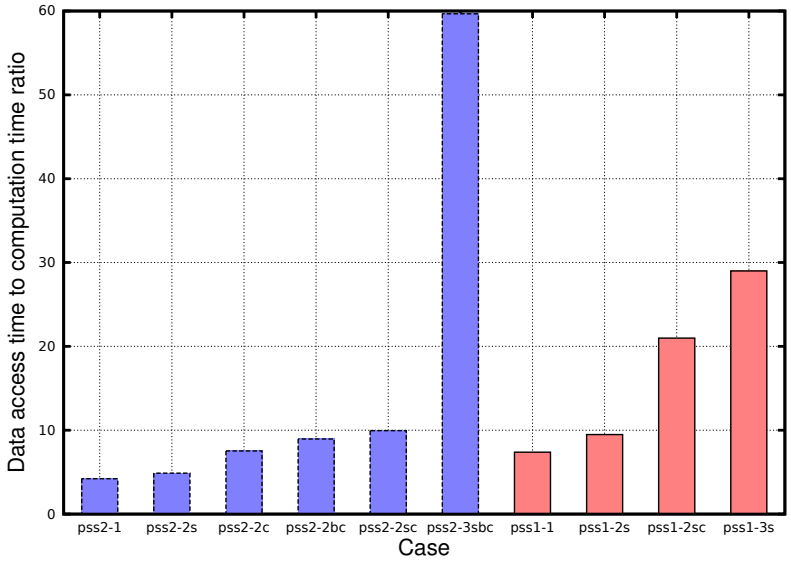


Figure 3.8: Y axis is the ratio of actual data access time to computation time for Twitter benchmark observed in our experiments. X axis is the case abbreviation further illustrated in Table 3.3.

3.7 Incorporate with Locality Sensitive Hashing (LSH)

Locality Sensitive Hashing (LSH) [35, 31] is an approximate similarity search technique that scales to both large and high-dimensional data sets. Its basic idea is to hash the records using several hash functions to ensure that similar records have much higher probability of collision in buckets than dissimilar records.

An LSH scheme has the following defining property:

Definition 3.7.1. Let $f_{sim}(\cdot, \cdot)$ be a given similarity function defined on the collection of objects D. A distribution on a family H of hash functions operating on D is a locality sensitive hashing scheme if for d_i, $d_j \in D$,

$$\mathbf{Prob}_{h \in H}[h(d_i) = h(d_j)] = f_{sim}(d_i, d_j).$$

Using this scheme, hash functions h_1, h_2, $\cdots$, h_m drawn from H are applied to raw vectors to encode them into signatures of hash values.

Before introducing our LSH approach, we first explore two well-known methods to generate signatures from document vectors: Min-hash [17] for Jaccard similarity and random projection [21] for cosine similarity.

Min-hash [17] is the min-wise independent permutations method used in Shingling. For each of the random orderings of features in a document vector, the

feature with lowest order is picked as the *minimum hash*. The probability that two documents d_i and d_j have the same min-hash feature for a given ordering is $\frac{d_i \cap d_j}{d_i \cup d_j}$(i.e., Jaccard similarity). This procedure is repeated for k different randomly selected orderings to reduce the risk of false positives; thus, the min-hash signature of a document vector consists of all k min-hash values.

Random projection [21] uses a series of random hyperplanes as hash functions to encode document vectors as fixed-size bit vectors. Assume there are in total m dimensions of features. To obtain a signature of k bits using this approach, k randomly generated real-valued vectors of length m are used to map each document vector d onto a signature $\in [0,1]^k$. The i^{th} bit is determined by an inner product of d and the i^{th} random vector r_i. The signature is computed as follows:

$$h_{r_i}(d) = \begin{cases} 1, & \text{if } r_i \cdot d \geq 0 \\ 0, & otherwise \end{cases}$$

The cosine similarity between two documents can be computed via hamming distance between their signatures, according to the following relation:

$$Sim(d_i, d_j) = cos[(\pi \cdot \frac{hamming(h(d_i), h(d_j))}{k})].$$

Previous work have applied variants of LSH on cross-language information retrieval problem [51] and near duplicate detection [33]. The work included in Ivory package [51] applies sliding window mechanism on sorted signatures of hamming

distances in the hash table generated by one set of hash functions, and repeat this step for hundreds of rounds. Due to errors introduced by bit signatures and sliding window algorithm, the upper bound of recall for their method is 0.76 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.

An adaptive approach [33] tunes LSH by concatenating k hash values from each data object 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 3.9. 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 by k.

We take a different angle in design, and opt for a relatively lower value of k and relatively higher value of l. More hashing rounds (l) contributes to a higher level of recall. The drop of number of hash functions (i.e. the number of hash values or bits generated in each round) will speed up the LSH process. After the centralized LSH step, we apply our efficient Partition-based algorithm in parallel upon all buckets generated in all rounds of LSH. Such pipeline is illustrated in Figure 3.10. Notice

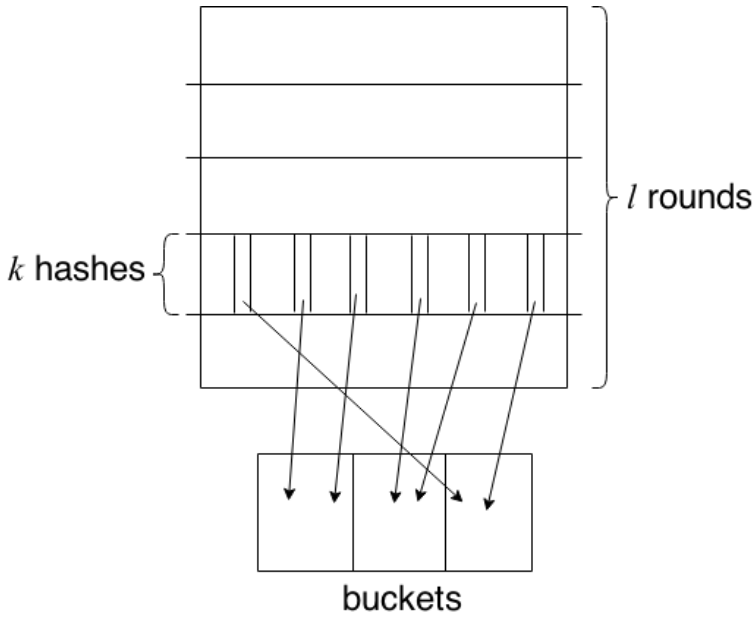


Figure 3.9: Illustration of l rounds of LSH, each round generates k hash values.

the LSH step is common before the original input data is copied and processed in parallel via PSS tasks. LSH computation is also parallelized over the distributed servers in the form of MapReduce jobs, and consists of sub-steps of projection generation, signature generation and bucket generation.

Our combined algorithm design achieves 100% precision with a guaranteed recall ratio. The reason is explained as below. In terms of hashing functions, we mainly apply the random projection algorithm for cosine similarity. Suppose the probability that two signature bits (at the same position) from two records collide equals to the cosine similarity of the two records. Given cosine similarity threshold τ, the number of bits for a signature k, and the number of LSH rounds

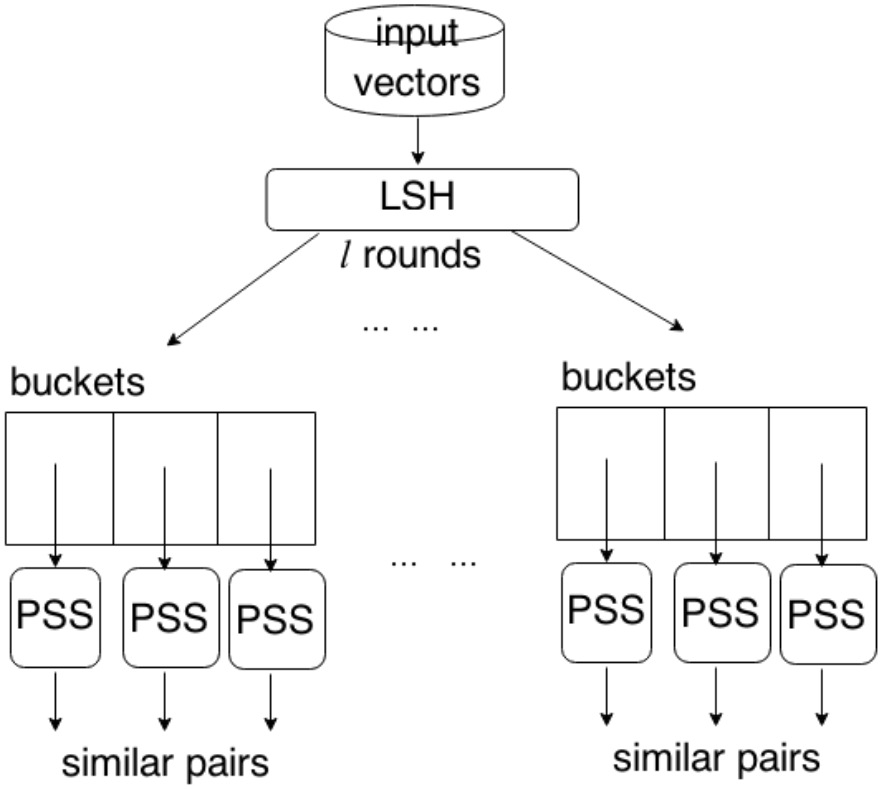


Figure 3.10: Our implementation of LSH and PSS pipeline.

Table 3.4: Number of LSH rounds l needed to achieve targeted recall rate $recall$ for cosine similarity threshold τ, given 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

l, the recall rate is as follows:

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

Based on this formula, we can compute the value of l, given cosine similarity threshold τ, the number of bits for a signature k, and a targeted recall rate $recall$:

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

Given various choices of signature bits k, targeted recall rate $recall$, and cosine similarity threshold τ, the corresponding rounds of LSH needed (l) are listed in Table 3.4.

Besides the high level of precision and recall rates, this combined approach is, in general, more efficient than the Partition-based similarity search (PSS) due to hashing. If we assume in each round, all records are evenly divided among buckets. Without considering the additional cost of generating LSH signatures and making copies of records to different buckets, we could reduce the total number of similarity computation to a fraction of $\frac{l}{2^k}$ of the original. This is because given n records, the number of pair-wise similarity computation 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, such ideal speedup ratio could never be reached. The actual speedup ratios are reported in Section 3.9.6. When applicable, we could even combine the LSH and PSS together and achieve higher level of speedup. The trade-off is also discussed in Section 3.9.6.

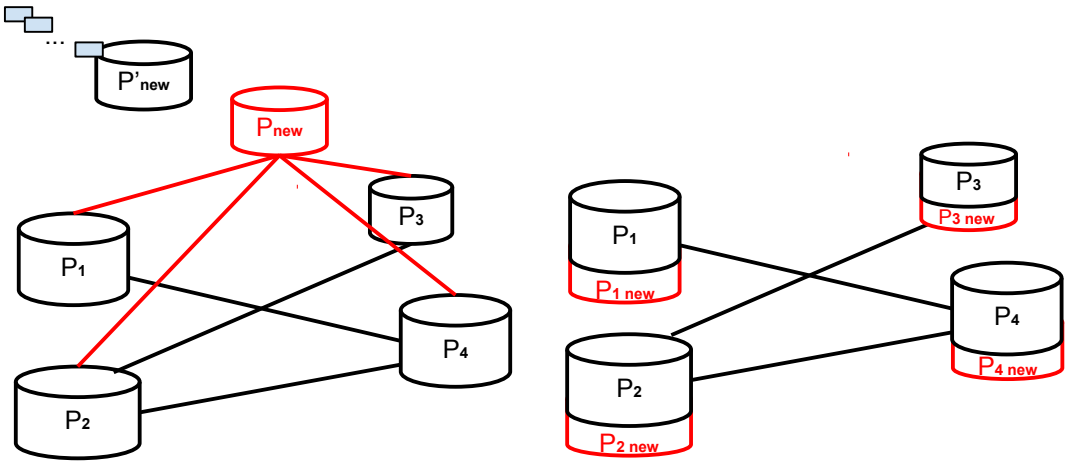
3.8 Discussions

We discuss some additional issues for our partitioned similarity search algorithms.

3.8.1 Partition-based Similarity Search with Incremental Updates

In various applications, content could be appended to the original set periodically. For example, web search engine constantly crawls the web for updated content, Twitter users continue creating new tweets, music website users sometimes update ratings or add new ratings to songs they listen to. How to handle incremental content update in Partition-based similarity search without sacrifice efficiency?

Instead of naïvely applies all-pairs similarity search over the whole universe of records, we set aside a *new* partition. Every time new documents and / or new versions of old documents are generated, we append them to the end of the *new* partition. Once the *new* partition has grown to a threshold size or a threshold amount of time is reached, we start a MapReduce job to compare the *new* partition with all the original partitions, similar to what we usually do for all the original partitions. After the comparison, this *new* partition is added to the original set as a stand-alone partition with potential similarity relationship to all the other partitions. And the following new or updated records could be appended to another *new* partition and repeat such a process as illustrated in Figure 3.3 (a).



(a) Pipeline for processing incremental updates.

(b) End result after updating static partitions with the *new* partition.

Figure 3.3: Incremental update illustrations.

Another issue worth mention is the update of static partitions once the comparison of records in new partition is completed. We explain as follows. Each updated document d_x could be inserted into group G_i if i is the minimum integer that satisfies $\|d_x\|_1 \leq \max_{d_y \in G_i} \|d_y\|_1$. This document d_x inserted in group G_i is further mapped to subgroup $G_{i,j}$ where j is the maximum integer satisfying $\max_{d_y \in G_j} \|d_y\|_1 < \frac{\tau}{\|d_x\|_\infty}$. Each chunk of updated documents is appended to the end of its corresponding partition and the partition size grows. Figure 3.3 (b) illustrates how these partitions look like after documents being appended based on aforementioned schema.

3.8.2 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 [50], it is easy to verify that if one of the following inequalities is true:

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

The static partitioning algorithm still sorts all vectors by norm $||d||_1$. 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 6 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 current maximum value $\|d_i \cdot d_j\|_1$. Term $score[i][j]$ in Lines 3 and 4 of Function 6(continued) 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:

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

Then the static partitioning algorithm can 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 6 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 3 and 4 of Function 6(continued) is changed with the following Dice similarity formula

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

3.8.3 Compare with 2D Blocking Strategy

We can view S_i and $\mathcal{B}$ as two matrices and PSS1 has used a row-wise block data layout for S_i while PSS2 adds a row-wise blocking in area $\mathcal{B}$. There is an extension option to further divide S_i and $\mathcal{B}$ as a set of sub-matrices, which can potentially further improve the use of cache temporal locality in both matrices. We call this extension 2D Blocking, where 2D stands for two-dimensions: both row-wise and column-wise. 2D Blocking follows the previous scientific computing research that views a sparse matrix as a collection of dense small sub-matrices and employs BLAS3 to perform sub-matrix multiplication [25, 48, 53]. However, our experimental results in Section 3.9.9 show that vector-feature matrices in the tested applications are extremely sparse and 2D Blocking does not contribute enough benefits to counteract the introduced overhead.

3.9 Evaluations

We have implemented our algorithms in Java. The source code and test datasets could be found at `https://github.com/ucsb-similarity/pss`.

Our evaluations have the following objectives:

1. Explain the problem complexity and demonstrate the execution scalability by reporting the speedup over the sequential time as we scale the number of cores utilized.

2. Compare our partition-based method with two alternative MapReduce solutions and assess the benefit of static partitioning.

3. 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.

4. Report the efficiency and effectiveness of incorporating LSH with PSS, and provide guideline for method choices that meet different requirement.

5. Evaluate the experimental results when a new partition is used during incremental updates.

6. Evaluate 2D Blocking to understand the issues of subm-atrix multiplication for APSS.

7. Compare the cache behavior and execution time for metrics other than Cosine: Jaccard and Dice. Discuss the results for both PSS1 and PSS2.

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 ClueWeb collection [9]. The average number of features is 320 per web page. We choose 40M records because it is already 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 stop-word filtering [41].

Environment setup. We ran parallel speedup experiments on a cluster of servers each with 4-core AMD Opteron 2218 2.6GHz processors and 8G memory and a cluster with Intel X5650 6-core 2.66GHz dual processors and 24GB of memory per node. We mainly report performance on the AMD cluster because a larger Intel cluster environment was less available for us to conduct experiments. The cache-conscious experiments were also conducted on 8-core 3.1GHz AMD Bulldozer FX8120 machines. Each AMD FX8120 processor has 16KB of L1d 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.

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%

Table 3.5: Cost of static partitioning and runtime cost distribution of PSS in parallel execution.

Table 3.5 shows that static partitioning which is also 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. This implies that the computation cost in APSS is dominating and hence load balance of the computation among cores is critical for overall performance.

The static partitioning step takes less than 3% of the total parallel execution time. The cost of self-comparison among vectors within a partition is included when reporting the actual cost.

To support our arithmetic models, we also provide empirical evidence measured by the Linux profiling tool *perf*. Perf collects the performance counters that count 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. The L1 caches is the most commonly accessed cache, and often have low associativity. The L3 cache has the most influence on runtime, as it masks accesses to main memory.

3.9.1 Problem Complexity and Scalability of PSS

Rows 3 and 4 of Table 3.6 list the sequential execution time in hours for Twitter, ClueWeb and YMusic datasets with different sizes when running PSS2. The values marked in gray are estimated by sampling part of its computation tasks, considering the fact that computation load grows quadratically as problem

Dataset	Twitter		ClueWeb		YMusic
Size	4M	100M	1M	40M	625K
AMD	45	45,157	50	79,845	31.95
Intel	26.7	25,438	29.3	46,946	17.8
AMD/df-limit	1.27	797	4.55	7,286	6.23

Table 3.6: Sequential time in hours on AMD Opteron 2218 2.6GHz and Intel X5650 2.66GHz processors (τ=0.8). The values marked in gray are estimated results based on sampling, due to time and resource constraint.

size grows. Such estimation is reasonably accurate since 4M Twitter data or 1M ClueWeb data is large enough to represent the data skewness, increasing the size by 10x merely enlarges the number of tasks and the workload of each tasks by the corresponding ratio. From the results in Rows 3 and 4, APSS is a time consuming process. Even for a Twitter dataset with 4M tweets, the entire dataset can fit in the memory; but it still takes a couple days to produce the results. Parallelization can shorten the job turnaround time and speedup iterative data analysis and experimentation.

Stop words are removed in the Twitter and ClueWeb input datasets; additional approximated preprocessing may be applied to reduce sequential time significantly if the trade-off in accuracy is acceptable [26, 41]. For example, the bottom row

of Table 3.6, marked as "df-limit", lists the sequential time on an AMD core after removing features with their vector frequency exceeding an upper limit proposed in [41]. After sampling a the ClueWeb dataset, 49 words with document frequency above 200,000 are excluded in web page comparison and the sequential time is shortened by 11x. Using this df-limit strategy reduces the sequential time by 35.3x or more for Twitter and by 5.1x for YMusic. In the rest of this section, we report performance of *exact* similarity search without using approximated pre-processing such as df-limit.

It should be emphasized that the algorithms discussed in this dissertation conduct *exact* similarity comparison without approximation, unless otherwise specified.

With exact similarity comparison, Figure 3.4 shows the speedup and parallel time for processing 40M ClueWeb dataset and 100M Twitter dataset when varying the number of cores. Due to the time constraint in our shared cluster environment, we report the average execution time of multiple runs after randomly selecting 10% of ClueWeb parallel tasks and 20% of Twitter tasks. Such a sampling methodology follows the one used in [41]. Speedup is defined as the sequential time of these tasks divided by the parallel time. 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%

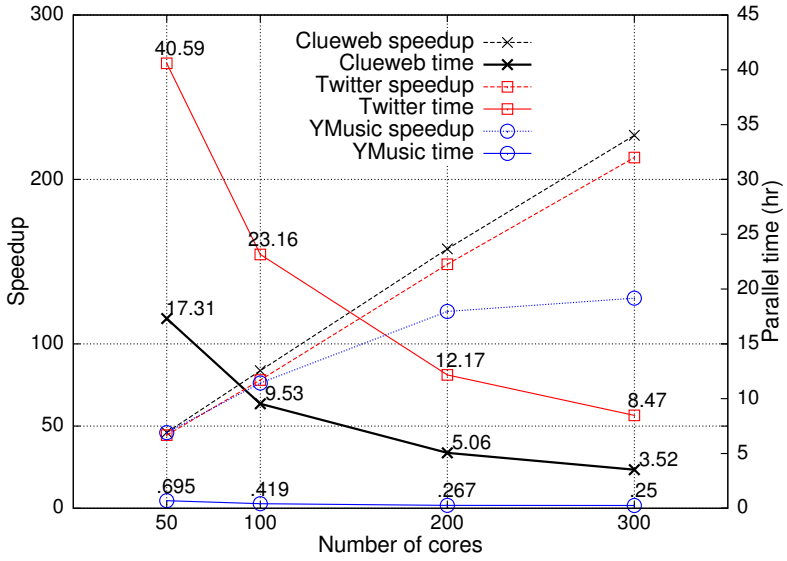


Figure 3.4: X axis is the number of cores used. Left Y axis is speedup. Right Y axis is parallel execution time for the selected tasks.

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 with 31.95 hour sequential time are 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 is not used in the scalability experiments due to similar reasons.

3.9.2 Comparative Studies

We also calculate the average time for comparing each pair of vectors normalized by their average length in a dataset. Namely $\frac{\text{Parallel time}\times\text{No of cores}}{\text{No of pairs}\times\text{Vector length}}$. The normalized pair-wise comparison time is about 1.24 nanoseconds for Twitter and 0.74 nanoseconds for ClueWeb using 300 AMD cores given $\tau = 0.8$. Varying the number of cores affects due to the difference in parallel efficiency. Varying τ also affects because it changes the results of dissimilarity-based partitioning and

graph structure. This number can become smaller if approximated preprocessing is adopted [26, 41].

To confirm the choice of partition-based search, we have also implemented an alternative MapReduce solution to exploit parallel score accumulation following the work of [41, 13] where each mapper computes partial scores and distributes them to reducers for score merging. The performance comparison is presented in Figure 3.5 The parallel score accumulation is much slower because of the communication overhead incurred in exploiting accumulation parallelism. 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.

As sanity check, we also estimate the normalized pair-wise comparison time reported in [41]. To compare 90K vectors with 4.59 million MEDLINE abstracts using at most 60 terms per vector on about 120 cores each with 2.8GHz CPU, it takes a MapReduce solution called PQ [41] 448 minutes with approximated preprocessing, meaning about 130.1 nanoseconds to compare each normalized vector pair, while PSS takes about 1.24 nanoseconds for Twitter and 0.74 nanoseconds for ClueWeb per normalized vector pair.

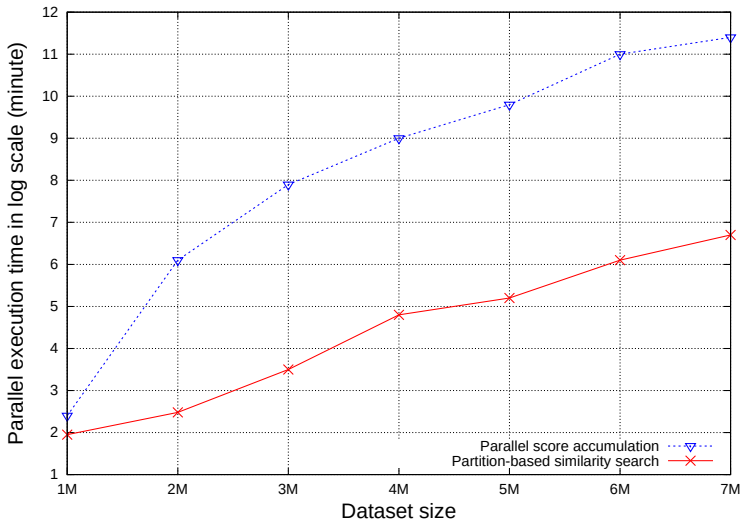


Figure 3.5: Parallel time on 120 cores of parallel score accumulation method and partition-based similarity search over Twitter dataset as data size increases from 1M to 7M ($\tau = 0.9$). Time is reported in log scale.

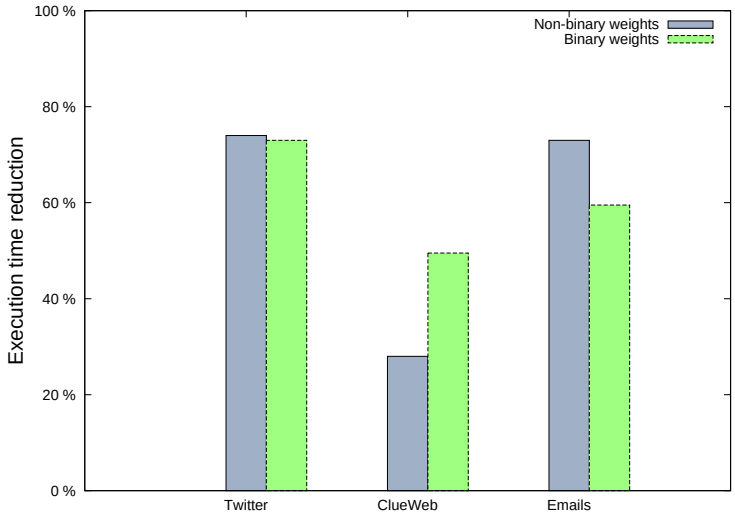


Figure 3.6: Percentage of execution time reduction after applying static partitioning using either original weights or binary weights.

Figure 3.6 demonstrates the effectiveness of static partitioning by showing the percentage of parallel execution time reduced after static partitioning is applied. The number of cores allocated is 120 for the Twitter and 10M of ClueWeb datasets, and 20 cores for the Emails dataset. Static partitioning with dissimilarity detection leads to about 74% reduction for Twitter, about 29% for ClueWeb, and about 73% for Emails dataset. We also gained similar results when binary feature weights are used.

3.9.3 Performance of PSS, PSS1 and PSS2

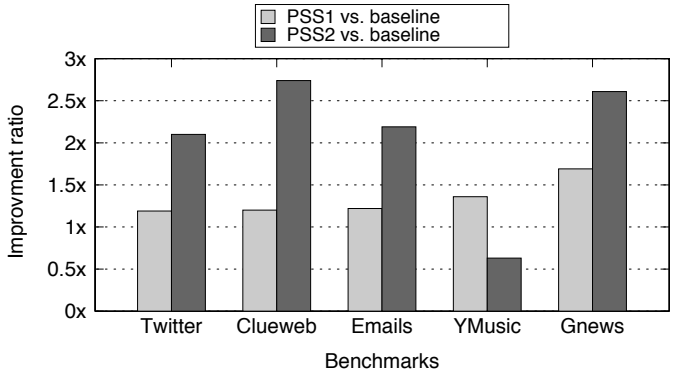
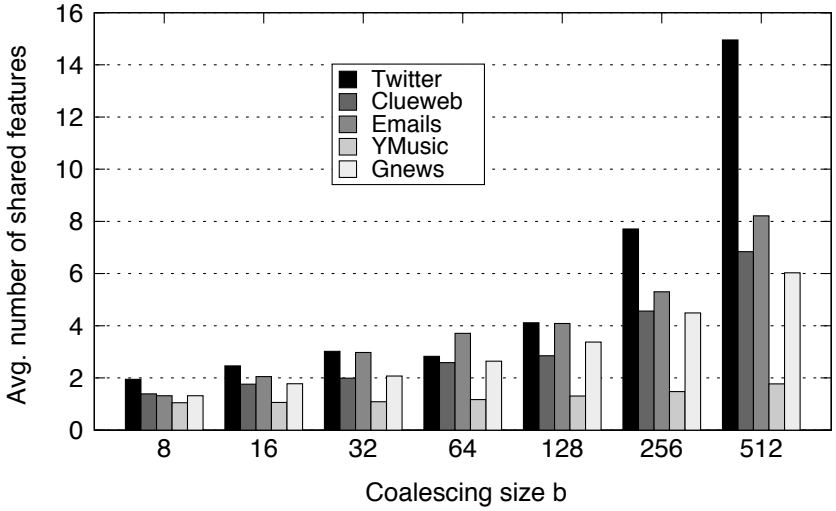


Figure 3.7: Y axis is improvement ratio $\frac{Time_{PSS}}{Time_{PSS1}}$ and $\frac{Time_{PSS}}{Time_{PSS2}}$. The average task running time includes I/O.

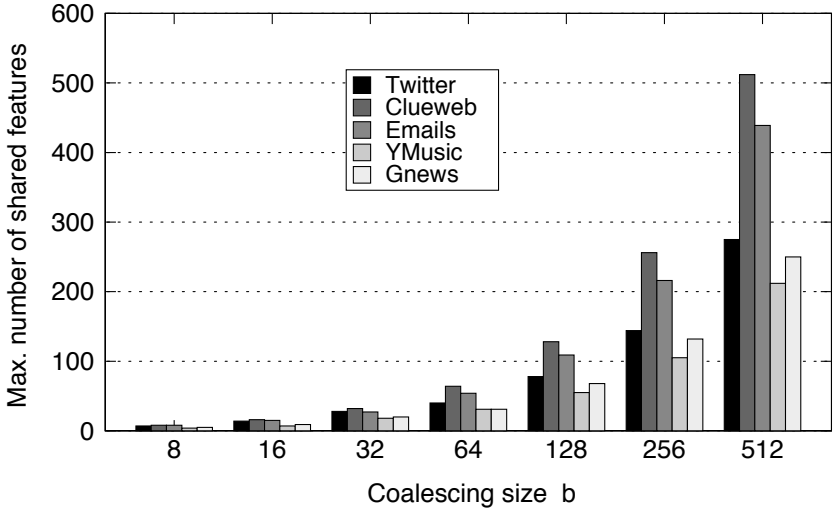
In the following subsections within this chapter, we mainly report and compare the running time for different algorithms when the static partitions are given.

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 3.7 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 3.8 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



(a) The average number of shared features among b vectors



(b) The maximum number of features shared among b vectors.

Figure 3.8: Number of features shared for five datasets.

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 in YMusic dataset.

3.9.4 Cache Behavior and Cost Modeling for PSS1

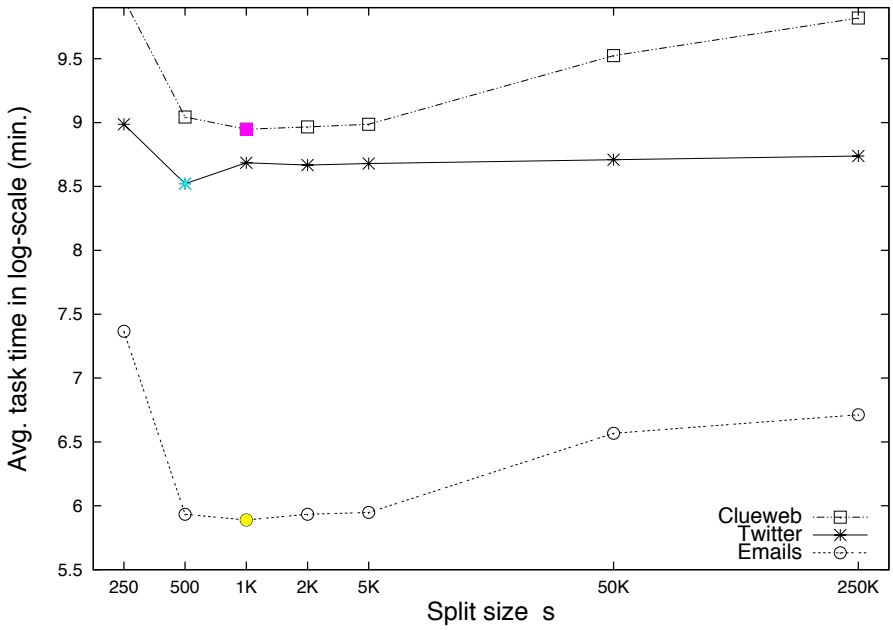


Figure 3.9: The average running time in *log scale* per PSS1 task under different values for split size s. The partition size S for each task is fixed, $S = s \times q$.

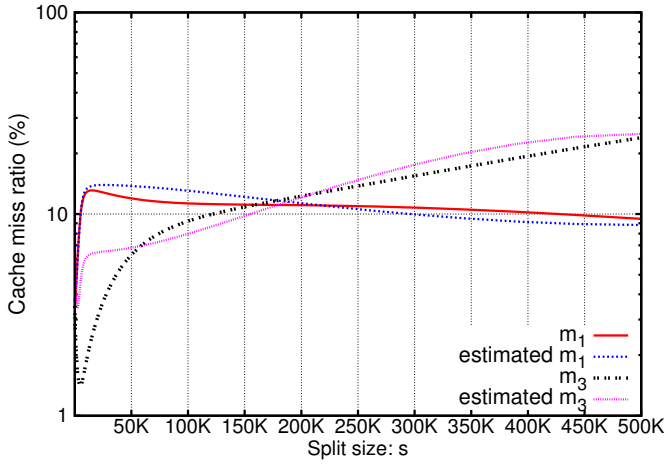
The gain from PSS to PSS1 is achieved by the splitting of the hosted partition data. Figure 3.9 shows the average running time of a PSS1 task including I/O in log-scale with different values of s. Notice that the partition size ($S = s \times q$)

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 running time is achieved when s value is ranged between 0.5K and 2K, consistent with our analytic results.

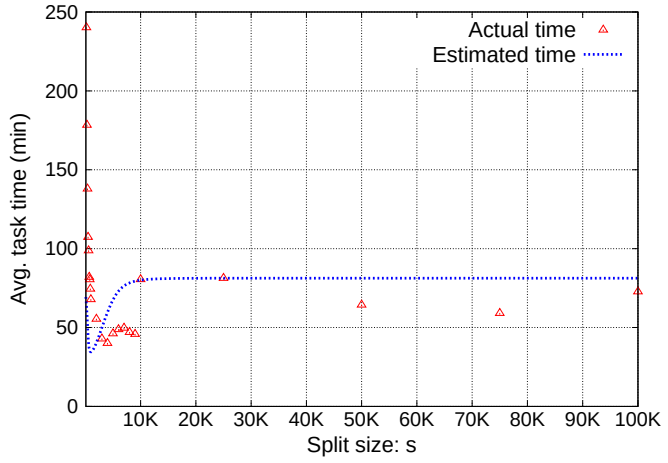
We demonstrate the cache behavior of PSS1 modeled in Section 3.4.2 with the Twitter dataset.

Figure 3.10(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% 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 3.10(b). Our estimation of cache miss ratios fits the real ratios quite well, and 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



(a)



(b)

Figure 3.10: 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.

too high and thus the actual performance is poor. When s ranges from 50K to 80K, the actual running time drops slightly. This is because as s increases, there is some benefit for amortizing the cost of inverted index look-up. Both the estimated and real time results suggest that the optimum s value is around 2K. Given the optimum s, PSS1 is at least twice faster than when s is 10K.

3.9.5 Impact of Parameters and Cache Behavior for PSS2

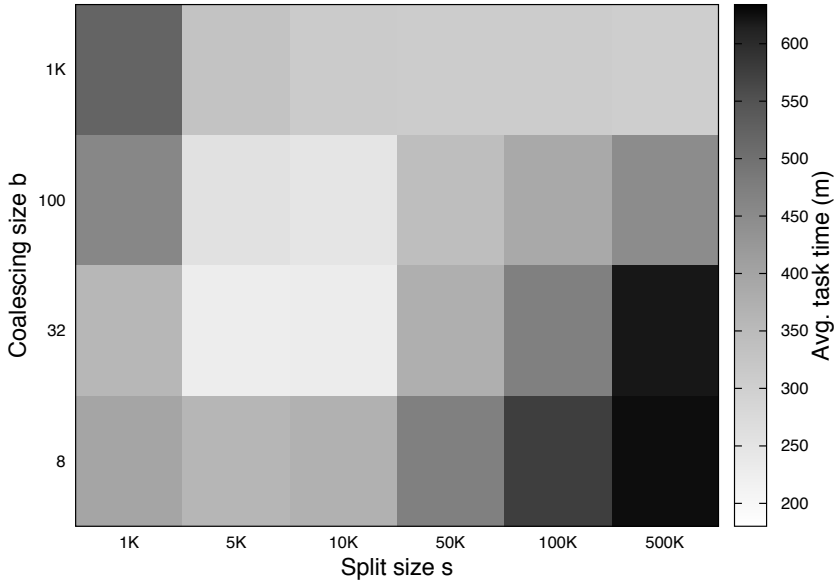


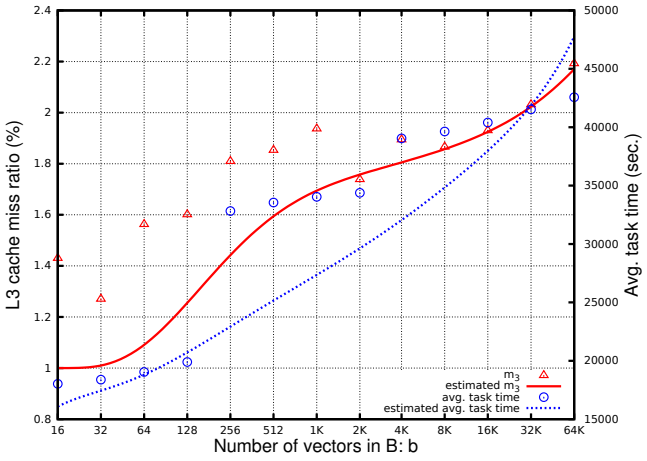
Figure 3.11: Each square is an $s \times b$ PSS2 implementation (where $\sum s = S$) shaded by its average task time for Twitter dataset. The lowest time has the lightest shade.

Table 3.7: Optimal parameters for PSS1 and PSS2 on AMD or Intel architecture.

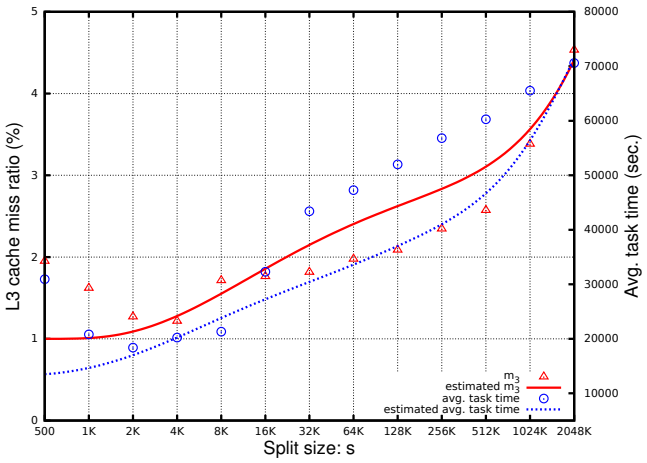
Architecture	Estimated			Actual		
	PSS1	PSS2		PSS1	PSS2	
	s	s	b	s	s	b
AMD	3,472	2,315	32	4,000	2,000	32
Intel	2,604	1,736	32	4,000	4,000	32

The gain of PSS2 over PSS1 is made by coalescing visits of vectors in B with a control. Figure 3.11 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 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.

Figure 3.12 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 analytic model correctly captured the trend and optimal values.



(a)



(b)

Figure 3.12: 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).

Table 3.7 lists the optimal parameters for PSS1 and PSS2 on AMD and Intel machines we have tested. As an example, we illustrate how to calculate the optimal parameters for PSS2 on AMD machines. As explained in Section 3.6, 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 parallel 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 an additional cache miss.

3.9.6 Incorporate with Locality Sensitive Hashing (LSH)

In order to reduce the computation complexity, we implement the LSH algorithm with random projection and apply it before PSS. In our implementation using Hadoop, first the LSH-related jobs run sequentially, including generating l

random projections each with k bits, generating signatures for all n records for l projections, generating buckets based on signature values, and prepare records for l rounds by copying records to bucket files for all l rounds. After the LSH mapping, every round starts a MapReduce job where each task in the job is responsible of conducting APSS for all records in this bucket (bucket-wise self-comparison). The LSH phase is conducted sequentially, while the l rounds of APSS are running in parallel.

Table 3.8 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 3.8: Runtime breakdown of conducting APSS for 20M Tweets with 95% target recall for all pairs with cosine similarity τ over 0.95 using 50 cores.

Also worth mention is that only applying LSH is not good enough, because it generates a very high number of false positives. This is due to the relatively small number of bits (k) we used in signature and the fact that the LSH rounds are treated with OR relation and the union of results are used. Table 3.9 compares our adopted method with two other approaches: running only LSH (Pure LSH) and running only PSS (Pure PSS). Pure LSH method with relatively high number of signature bits (k) could provide higher than 95% 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 $8.8x$ as much time as our adopted method which applies LSH before PSS. Such comparison shows the efficiency of conducting LSH before PSS

to speedup the process with bounded recall rate; and the necessity of conducting PSS after the LSH step as validation to ensure 100% precision.

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 3.9: 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 3.10 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. Due to the higher feature count per record and longer posting length in ClueWeb dataset, such a balance is achieved with a higher rounds of LSH k. 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 Table 3.6) 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 3.4. 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 3.11 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 3.10: 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 3.11: 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 [51] package applies sliding window mechanism on sorted signatures in order to reduce search space, but introduces 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 $\tau = 0.3$ [51]. 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 3.12. 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 3.12: A guideline for method choices that meet different requirement of target recall rate, target precision rate for a certain similarity threshold τ.

3.9.7 Incremental Updates

This subsection reports the efficiency of our algorithm when there is incremental content update. A naïve solution triggers a all-partition pairs comparison once a threshold is reached. Our method takes a more efficient approach. We set the threshold size as the median size of partitions. Once the new partition grows over the threshold size, a MapReduce job is started to compare only the new partition with all the original partitions. We compare our method of appending to a new partition (explained in Section 3.8.1) with the naïve solution. Table 3.13 shows that our approach is $50x$ faster than the naïve approach for similarity comparison of $100K$ Tweets update to an original set of $20M$ Tweets using 300 cores.

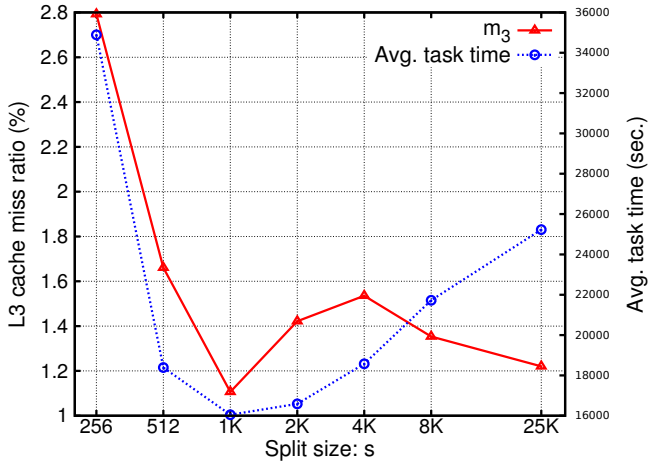
Initial size	Update ratio	Naïve method	Our approach
$20M$ records	0.5%	510 minutes	10 minutes
$20M$ records	5%	558 minutes	57 minutes

Table 3.13: 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.

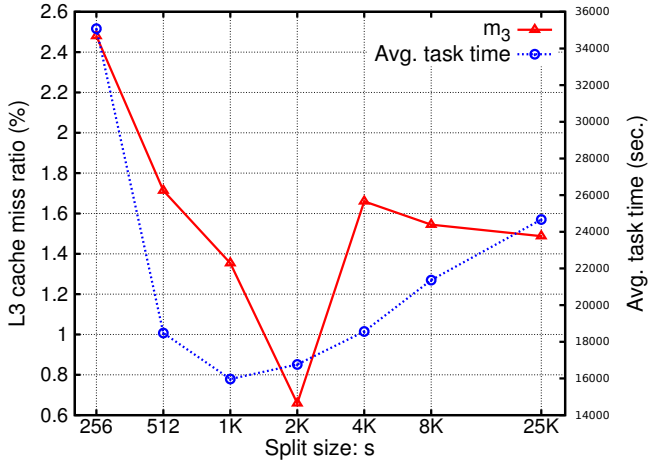
3.9.8 Similarity Measures

We assess the modified PSS1 and PSS2 in handling Jaccard and Dice metrics. Figure 3.13 shows how the average running time and L3 cache miss ratios change when different similarity measures are applied using PSS1. The trend and the extreme values (optimum s) are close despite the variety of similarity coefficients applied. The average task time for Jaccard and Dice coefficient are shorter than that of cosine, due to binary weights used. With 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 3.14 displays the contour graphs for L3 Cache 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

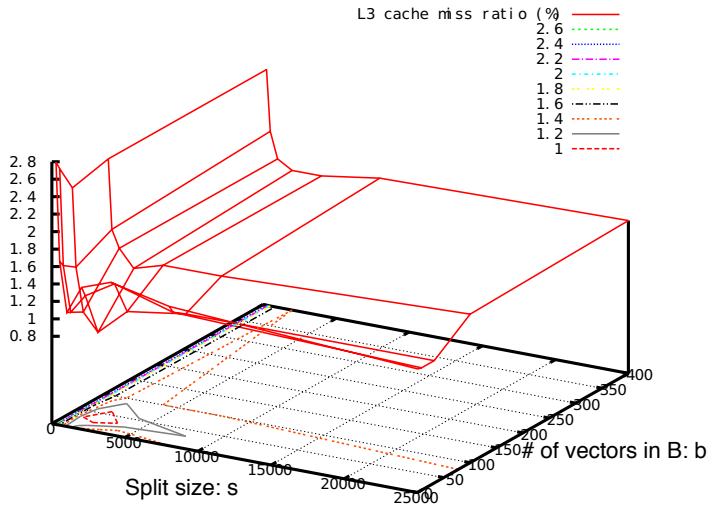


(a) Jaccard coefficient

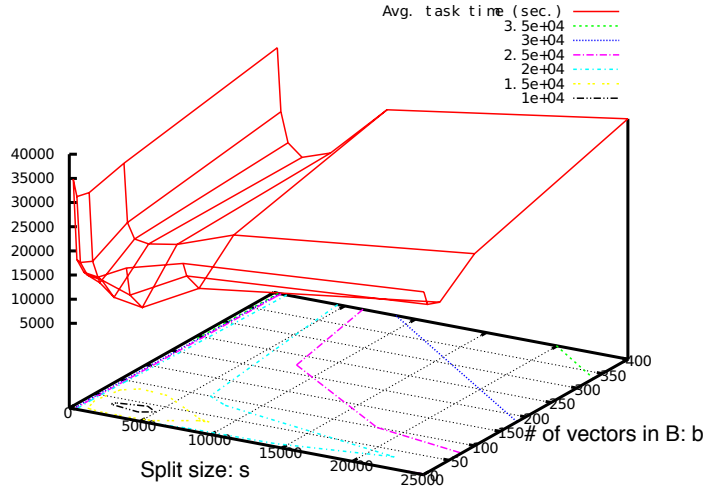


(b) Dice coefficient

Figure 3.13: 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 ($s \times q = 200K$).



(a)



(b)

Figure 3.14: 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.

4000 and b is around 32. The running time lasts as much as 3x longer when either s or b are chosen as values either too large or too small. We also observe a similar trend in the change of L3 cache miss ratio. Both mathematical analysis and experimental results show that our theory on the cache-guided parameter choices of PSS1 and PSS2 algorithms could not only be applied to cosine similarity metric, but to other similarity measures as well.

3.9.9 A comparison with 2D Blocking

We assess the individual task performance in utilizing the CPU resource by collecting its mega-flops rate and compare it with the peak mega-flops rate when vectors are dense. Similarity computation can be viewed approximately as a sparse matrix multiplication together with dynamic computation filtering. We assess the gap between how fast each CPU core can do in terms of peak application performance with a dense matrix and what our scheme has accomplished. First we compare the mega-flops performance of our Java code with MTJ [34] from Netlib, which is highly optimized for dense matrix multiplication. The mega-flops achieved by a dense matrix multiplication routine (called dgemm) in MTJ achieves 1500 mega-flops for matrix dimension 1000 on a single core and achieves 500 mega-flops for a small dense matrix. Our scheme achieves 280 mega-flops for

Twitter benchmark. That is fairly high considering we are dealing with extremely sparse matrices.

In 2D Blocking design, we represent feature vectors in S and B as a set of small dense sub-matrices and employ a built-in MTJ BLAS3 dense matrix routine to multiply these sub-matrices. The advantage of 2D Blocking is that we leverage MTJ, a highly optimized library for cache performance. The disadvantage is that these small dense matrices still contain many zeros and a BLAS3 routine does not remove the unnecessary computation operations as well as an inverted index does. Figure 3.15 lists the comparison between 2D Blocking and PSS2 performance, with the ratio $\frac{Time_{2DBlocking}}{Time_{PSS2}}$ for different block settings. 2D Blocking is unfortunately much slower than PSS2. The reason is that vector-feature matrices in the tested similarity applications are extremely sparse and the 2D Blocking strategy with BLAS3 does not contribute enough benefit to counteract the introduced overhead.

Table 3.14 provides another angle to explain why 2D Blocking slows down the task. We list the average fill-in ratio of those nonzero sub-matrices handled by 2D Blocking. Fill-in ratio is the number of stored values which are in fact zero divided by the number of true non-zeros. The fill-in ratio is very high in our tested benchmarks, and the number of true non-zeros for each block is too low to gain enough benefit with such blocked approach.

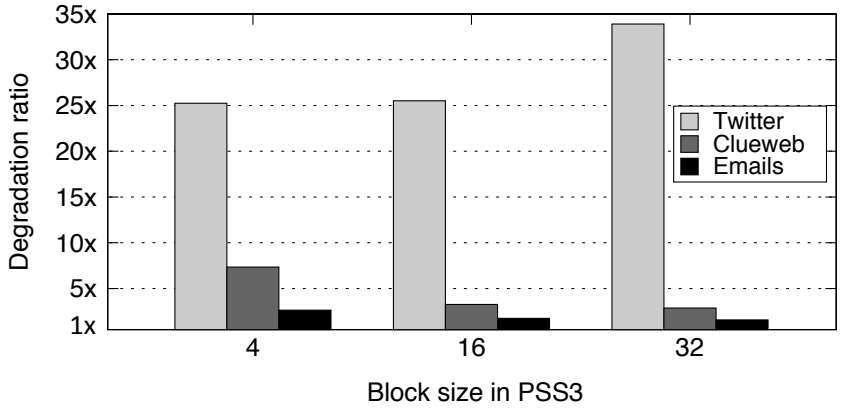


Figure 3.15: Y axis is ratio $\frac{Time_{2DBlocking}}{Time_{PSS2}}$. X axis is different block sizes used in 2D Blocking algorithm when compared with PSS2. 2D Blocking is slower than PSS2 in general under different blocking sizes.

Block size	**4×4**	**4×8**	**4×16**	**16×16**	**32×8**	**32×16**
Twitter	2.5	3.7	3.9	6.2	5.3	7.7
ClueWeb	2.6	8.2	4.8	5.6	4.4	6.2

Table 3.14: Average fill-in ratio with different block sizes.

Chapter 4

Load Balance for Partition-based Similarity Search

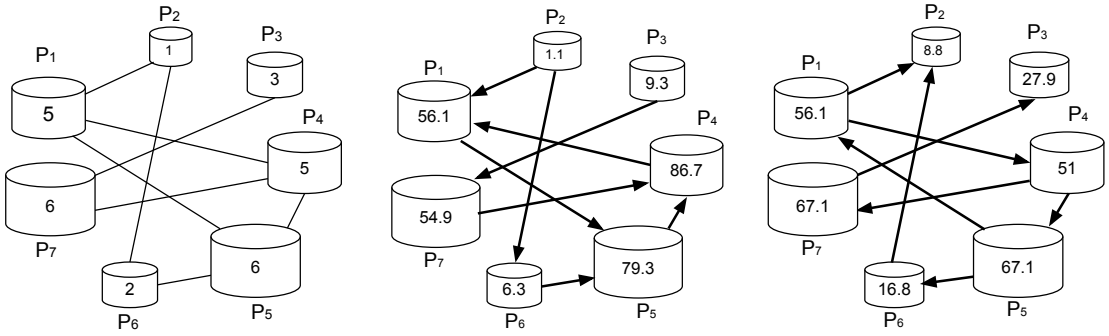
4.1 Load Balance Problem

We formalize the load assignment problem as follows. The data partitioning phase defines a set of v partitions and their potentially similar relationship. This can be represented as a graph, called a similarity graph defined next.

Definition 4.1.1. Similarity graph (G): Let G be an undirected graph where each node represents a data partition and each edge indicates potential similarity relationship between the two partitions it connects.

Since the similarity result of two vectors is symmetric, comparison between two partitions P_i and P_j should be only conducted by one of the corresponding tasks T_i or T_j. A load assignment algorithm determines which task performs this comparison. The load assignment process converts the undirected similarity graph into a directed graph in which the direction of each edge indicates which task conducts the corresponding comparison. We call this a comparison graph and it is defined as follows.

Definition 4.1.2. Comparison graph (D): Let D be a directed graph where each node represents a data partition. An edge $e_{i,j}$ from partition P_i to P_j indicates that task T_j compares P_j with P_i.



(a) (b) (c)

Figure 4.1: (a) An undirected similarity graph; node weights are partition sizes. (b) A directed comparison graph for (a); node weights are the corresponding task cost. (c) Another comparison graph for (a).

Comparison graph D contains the same set of nodes and edges as the corresponding similarity graph G, except that the edges in D are directed. The directed edges reveal the data flow direction when comparing two potentially similar partitions. Figure 4.1(a) illustrates a similarity graph with seven nodes. P_1 is potentially similar to P_2, P_4 and P_5, for instance. The comparison between P_1 and P_2 can be performed by either T_1 or T_2. The numbers marked inside the graph nodes are partition sizes, proportional to the number of vectors in the partition. Figures 4.1(b) and 4.1(c) show two comparison graphs with different load assignments. The number marked inside a comparison graph node is the corresponding task cost and we explain the cost model below.

The cost function of each task consists of computation cost and data I/O cost. For each task defined in Algorithm 1, the computation cost includes the cost of an inverted index look-up, multiplication and addition, and memory/cache accesses. While a thorough cost model involves memory hierarchy analysis [4], the overall computation cost can be approximated as proportional to the size of the corresponding partition P_i multiplied by the size of the potentially similar partitions to be compared with. The data I/O cost occurs when fetching P_i and other partitions from local or remote machines, and also when storing the detected similarity results on disk. Since the start-up I/O cost and transmission bandwidth difference to the local or remote storage are relatively small, the I/O

cost is approximately proportional to the size of the partitions involved. Note that the runtime scheduling that maps tasks to machines is affected by data locality. As we discuss later, the computation cost is dominating in APSS and thus the I/O cost difference caused by data locality is not sufficient enough to alter our optimization results in terms of competitiveness to the optimum.

Define the cost of task T_i corresponding to partition P_i in comparison graph D as:

$$Cost(T_i) = f(P_i, P_i) + f_c(P_i) + \sum_{e_{j,i} \in D} (f(P_i, P_j) + f_c(P_j))$$

where $f(P_i, P_i)$ is the self comparison cost for partition i and is quadratically proportional to the size of P_i. $f(P_i, P_j)$ is the comparison cost between partition i and j. It satisfies that $f(P_i, P_j) = f(P_j, P_i)$ and this cost is proportional to the size of P_i multiplied by size of P_j. $f_c(P_i)$ is the I/O and communication cost to fetch partition P_i from local and/or remote storage and output the results of self-comparison. $f_c(P_j)$ is the cost to fetch partition P_j and output the similar pairs between P_i and P_j. For Figures 4.1(b) and 4.1(c), $f(P_i, P_j)$ is a multiplication of the sizes of P_i and P_j, and $f_c(P_i)$ is estimated as 10% of the size of P_i. In Figure 4.1(c), $Cost(T_5)$=67.1 because $f(P_5, P_5)$=36, $f(P_5, P_4)$=30, $f_c(P_5)$=0.6 and $f_c(P_4)$=0.5.

Different edge direction assignments can lead to a large variation in task weights. Let $Cost(D) = \max_{P_i \in D} Cost(T_i)$. For example, in Figure 4.1(b) $Cost(D)$

= 86.7 based on $Cost(T_4)$. In Figure 4.1(c) $Cost(D)$=67.1. Deriving a comparison graph that minimizes the maximum cost among all tasks is a key strategy in our design. As the load is shifted from the heaviest task to the other tasks, better load balancing is achieved.

A circular mapping solution in [5] compares a partition with half of other partitions, if they are potentially similar. When the number of partitions is odd, task T_i compares P_i with partitions P_j where j belongs to the set: $i\%v+1,\ (i+1)\%v+1,\ \cdots,\ (i+\frac{v-3}{2})\%v+1$. Figure 4.1(b) shows the circular solution for the similarity graph in Figure 4.1(a). T_1 is assigned to compare with partitions from P_2 to P_4, hence the edge is directed from P_2 and P_4 to P_1. Similarly, the comparison between P_1 and P_5 is assigned to P_5. The circular approach is reasonable when the distribution of node connectivity and partition sizes is not skewed. In practice, that is often not true.

Table 4.1 shows the variance of partition sizes and task costs in three datasets. The largest partition size could be many times larger than the average partition size and the standard deviation compared to the average size is also high. Additionally, the similarity relationship among partitions is highly irregular. Some partitions have lots of edges in similarity graph while others have sparse connections. Circular load assignment treats all partitions equally regardless of such variations and as a result, a task could be assigned all the comparison loads while

Dataset	Partition size (# of records per partition)			Task cost
	Avg	Std. Dev/Avg	Max/Avg	Max/Avg
Twitter	143,042	1.75	6.85	2.14
ClueWeb	337,720	0.67	2.37	4.25
YMusic	21,550	0.82	4.35	8.97

Table 4.1: Distribution statistics for partition size and parallel execution time with circular load assignment.

its counterpart tasks are very light. Column 5 of Table 4.1 shows the maximum divided by average task cost using circular assignment.

The ultimate goal of load assignment is to schedule computation to parallel machines with minimum job completion time. Since undirected edges in a similarity graph creates uncertainty in task workload, the key question here is what to optimize. Will balancing the task costs computed from the comparison graph help speedup the runtime execution without knowing the allocated computing resource in advance? In the next section, we discuss our optimization strategy and present a two-stage assignment algorithm.

4.2 Two Stage Load Balance Algorithm

Our algorithm for load assignment consists of two stages to derive a comparison graph with balanced load among tasks. The design considers uneven partition sizes and irregular dissimilarity relationship. The derived tasks are scheduled at runtime to q cores and the tasks with reduced variation in sizes contribute to better performance after scheduling. We will show that such a strategy can produce a solution competitive to the optimal solution for scheduling a similarity graph on a given number of cores. We discuss the two-stage algorithm in the following two subsections.

4.2.1 Stage 1: Initial Load Assignment

The purpose of Stage 1 of this algorithm is to produce an initial load assignment such that tasks with small partitions conduct more comparisons. This stage performs v steps where v is the total number of partitions in the given similarity graph. Each step identifies a partition, determines the direction of its similarity edges, and adds this partition along with these directed edges to comparison graph.

More specifically, each step works on a sub-graph of the original undirected graph G, called G_k at step k. G_1 is the original graph G. At step k, the algorithm identifies partition P_x with the lowest *potential computation weight* (PW). The

potential computation weight for task T_x based on sub-graph G_k is defined as:

$$PW(G_k, P_x) = f(P_x, P_x) + \sum_{e_{x,y} \in G_k} f(P_x, P_y).$$

It represents the largest possible computation weight for task T_x given the undirected edges in G_k. G_{k+1} is derived from G_k by removing the selected partition P_x and its edges in G_k. These edges connecting P_x in G_k are chosen to point to P_x in the generated directed graph.

Node	Init	Step 1	Step 2
	G_1	G_2	G_3
P_1	85	80	80
P_2	8	-	-
P_3	37	37	37
P_4	110	110	110
P_5	108	108	96
P_6	18	16	-
P_7	84	84	84

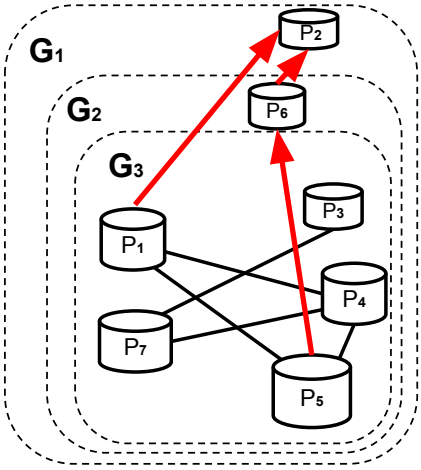


Figure 4.2: The first two steps in Stage 1 in the right figure, along with the PW values in the left table.

Figure 4.2 illustrates the first two steps in Stage 1. The left part of the figure lists the initial PW values of each node, as well as the corresponding values after the first step and second step. Partition P_2 has the lowest PW value initially and

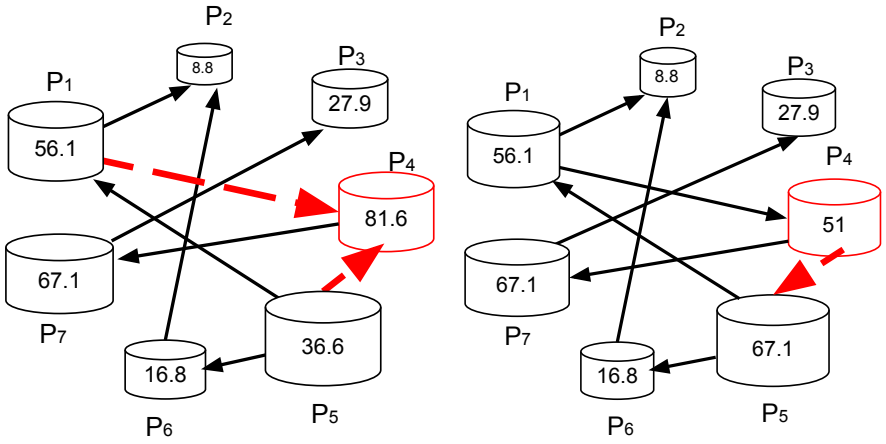


Figure 4.3: (a) The assignment produced in Stage 1. (b) The first refinement step in Stage 2: reversing edge $e_{5,4}$ to $e_{4,5}$.

is selected at Step 1. Edges connecting P_2 are all directed to P_2 in the formed directed graph. The PW values of the partitions adjacent to P_2 are changed from G_1 to G_2. Step 2 identifies P_6 as the the lowest PW in G_2, removing it and its edges from G_2. Finally the outcome of Stage 1 produces a comparison graph shown in Figure 4.3(a).

The cost of a task at Step k is considered to be *determined* if its corresponding partition has been selected before Step k. Otherwise, a task has a *potential* cost that equals to PW value plus possible I/O cost. Figure 4.4 shows the standard deviation of task costs at the first 200 steps using $Cost(T_i)$ if this task is determined, or its potential computation weight PW if it is undetermined. The

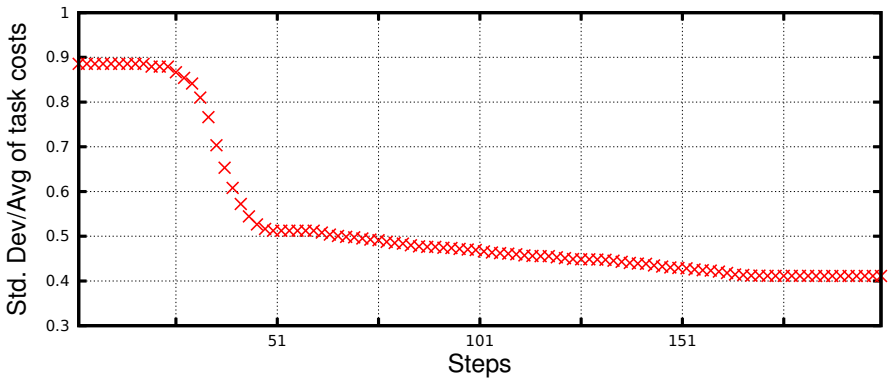


Figure 4.4: Monotonic decrease of the cost standard deviation in the first 200 steps in Stage 1 for Twitter dataset. The values are normalized by the average task computation cost.

step-wise trend illustrates that Stage 1 gradually reduces the variation of task costs.

Stage 1 pushes the computation load to the tasks with potentially low weight. This technique works better when partitions have highly skewed sizes since the lightest partitions absorb as much workload as possible. However, this greedy heuristic may cause some tasks to carry an excessive amount of computation. Another issue is that Stage 1 does not consider data I/O and communication cost, so the effect of optimization might be weakened. Hence, we introduce Stage 2 to

further refine the assignment produced by Stage 1 and mitigate the aforementioned weakness.

4.2.2 Stage 2: Assignment Refinement

Stage 2 conducts a number of refinement steps to reduce the load of the heavy tasks by gradually shifting part of their computation to their lightest neighbors. It performs the following procedure:

1. Find the task with the highest assigned cost $Cost(T_x)$. Identify one of P_x's incoming neighbors, say P_y, with the lowest cost among these neighbors, and reverse the direction of this edge from $e_{y,x}$ to $e_{x,y}$. Such a reversion causes a cost increase for T_y and a cost decrease for T_x. However, if the new cost of T_y becomes the same or larger than the original cost of T_x, this edge reversion is rejected. When an edge reversion is rejected, we continue with the incoming neighbor that has the second lowest cost. Repeat this process until a suitable neighbor is found so that the edge reversion successfully reduces $Cost(T_x)$. If all incoming neighbors of P_x are probed but no flip reduces $Cost(T_x)$ successfully, mark $Cost(T_x)$ as non-reducible.

2. Repeat the above step for the task with the highest weight after the update. If such a task is non-reducible, try the reducible task with the next highest

weight. If all nodes are marked non-reducible or the number of iterations tried reaches a predefined limit, the algorithm stops.

Figure 4.3(b) depicts the first refinement upon the output of Stage 1. The first edge probed in Figure 4.3(a) is $e_{5,4}$ because T_4 has the highest cost and T_5 has the lowest cost among all incoming neighbors of P_4 (i.e. P_1 and P_5). The reversion of edge $e_{5,4}$ to $e_{4,5}$ reduces $Cost(T_4)$ from 81.6 to 51 and boosts T_5 to be the task with the highest assigned weight, ready for the next probe. Since the flip of any incoming edge to P_5 does not further reduce $Cost(T_5)$, we do not flip. Finally, Stage 2 produces a comparison graph as shown in Figure 4.1(c) with $Cost(D)$=67.1.

4.3 Competitiveness Analysis

We do not know how the optimum scheduling solution dynamically maps tasks to machines at runtime as shown in Figure 4.6. However, we can use a bound analysis to show that our heuristic approach performs competitively in a constant factor compared to the optimum. We first address the load balancing issue without awareness of the machine location. Network distances impact the I/O and communication cost, but this cost is relatively less significant compared to

computation load imbalance in PSS. Define

$$\delta = \max_{P_i \in G}(\frac{f_c(P_i)}{f(P_i, P_i)}, \max_{e_{j,i} \in G} \frac{f_c(P_j)}{f(P_i, P_j)}).$$

This ratio represents the overhead ratio of I/O and communication involved in each task compared to its computation. In our experiments as shown in Table 3.5, I/O overhead is relatively small. Given this computation-dominating setting, for a cluster of machines with multiple CPU cores, we will simply view that the whole cluster has q cores without differentiating their machine location. The overhead in accessing data locally or remotely is captured in ratio δ.

Theorem 1 shows the result of two-stage load assignment algorithm is competitive to the smallest possible cost without knowing the number of cores available. Theorems 2 and 3 characterize the competitiveness of the algorithm to the optimum when the similarity graph is scheduled to q cores. The theorem proofs are listed in the appendix.

Theorem 4.3.1. *Define $Cost_{min}(G)$ as the smallest cost of a comparison graph derived from a given similarity graph G. The two-stage load assignment algorithm produces a comparison graph D with $Cost(D)$ competitive to $Cost_{min}(G)$. Their relative ratio satisfies*

$$Cost(D) \leq 2(1 + \delta)Cost_{min}(G).$$

Proof. Let $Cost_1(D)$ be the value of $Cost(D)$ after Stage 1. Refinements in Stage 2 do not increase $Cost(D)$ and thus $Cost(D) \leq Cost_1(D)$. We just need to show that Stage 1 can reach a solution competitive to $Cost_{min}(G)$. Namely $Cost_1(D) \leq 2(1+\delta)Cost_{min}(G)$.

Let D_i be a directed graph with all nodes $\in G_i$ and all edge orientations determined through the steps from G_i to G_{v-1} in stage 1, given a total of v partitions and D_1=D, G_1=G.

We use an induction to prove this theorem. The induction goes from D_{v-1} to D_1, reversing to the creation process in Stage 1. Towards the end of Stage 1, sub-graph G_{v-1} has two nodes left, and at most one edge between them. Choosing the partition with the smaller computation weight to perform the inter-partition comparison will add some communication and I/O cost, but leads to the balanced solution in this special case. Thus $Cost_1(D_{v-1}) = Cost_{min}(G_{v-1})$.

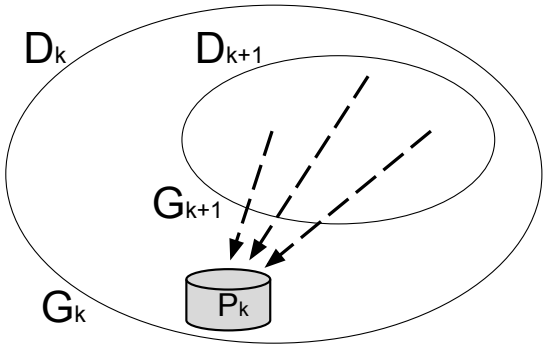


Figure 4.5: Illustration of D_k and D_{k+1} for induction proof.

Our induction assumption is that the solution for sub-graph D_{k+1} is competitive. Namely $Cost_1(D_{k+1}) \leq 2(1+\delta)\ Cost_{min}(G_{k+1})$. We want to show the solution for D_k is also competitive. Figure 4.5 illustrates sub-graphs D_k and D_{k+1}. Note that sub-graph D_k and G_k both have $v-k+1$ nodes and without loss of generality, these partition nodes are called $P_k, P_{k+1}, \cdots, P_v$. $Cost_{min}(G_k)$ satisfies

$$\begin{aligned} Cost_{min}(G_k) &\geq \frac{\sum_{j=k}^{v} f(P_j, P_j) + \sum_{k \leq i < j \leq v, e_{i,j} \in G_k} f(P_i, P_j)}{v-k+1} \\ &= \frac{\sum_{j=k}^{v} f(P_j, P_j) + \sum_{j=k}^{v} PW(G_k, P_j)}{2(v-k+1)} \\ &> \frac{\sum_{j=k}^{v} PW(G_k, P_j)}{2(v-k+1)} \\ &\geq \frac{(v-k+1)PW(G_k, P_k)}{2(v-k+1)} \\ &= \frac{1}{2} PW(G_k, P_k). \end{aligned}$$

Also notice that graph G_{k+1} is a sub-graph of G_k, then

$$Cost_{min}(G_k) \geq Cost_{min}(G_{k+1}).$$

Also following the definition of δ and the setting of $Cost(T_k)$ in Stage 1 of two-stage load assignment,

$$Cost(T_k) \leq PW(G_k, P_k)(1+\delta).$$

With the induction assumption and the above three inequalities, the outcome of Stage 1 with respect to D_k satisfies

$$\begin{aligned} Cost_1(D_k) &= \max\{Cost_1(D_{k+1}), Cost(T_k)\} \\ &\leq \max\{2(1+\delta)Cost_{min}(G_{k+1}), PW(G_k, P_k)(1+\delta)\} \\ &\leq (1+\delta)\max\{2Cost_{min}(G_k), 2Cost_{min}(G_k)\} \\ &= 2(1+\delta)Cost_{min}(G_k). \end{aligned}$$

Therefore

$$Cost(D) \leq Cost_1(D) = Cost_1(D_1) \leq 2(1+\delta)Cost_{min}(G).$$

□

The above result shows that the tasks produced by the two-stage algorithm have a fairly balanced cost distribution. As illustrated in Figure 4.6, a simple runtime scheduling heuristic is to assign tasks to idle computing units whenever they become available [29]. For example, the Hadoop MapReduce [23] scheduler works by assigning ready tasks in a greedy fashion with the best effort of preserving data locality. Once the central job tracker detects the availability of a task tracker, it assigns a ready task to the task tracker as long as there exists an unassigned task. When deciding which task to assign, it favors the tasks processing data local to or close to the machine of the task tracker. What is the performance behavior of our comparison tasks scheduled under such a greedy policy?

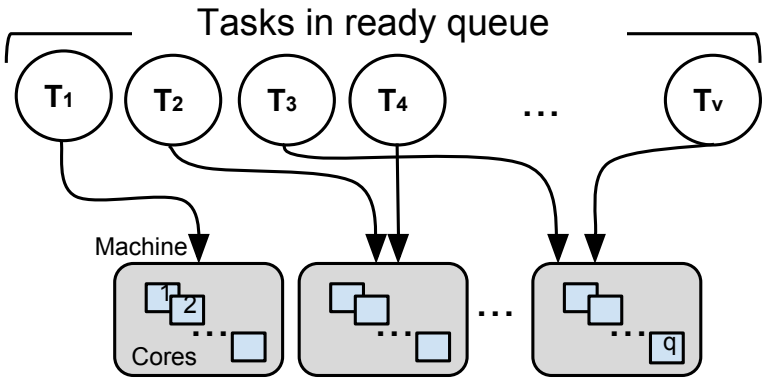


Figure 4.6: Greedy execution of v tasks at runtime on a cluster of machines with q cores.

The next theorem shows that under a greedy scheduler, the tasks produced by the two-stage algorithm perform competitively compared to an optimum solution.

Theorem 4.3.2. *The two-stage load assignment with a greedy scheduler produces a solution with job completion time PT_q competitive to the optimal solution with completion time PT_{opt}. Their relative ratio for dedicated q cores satisfies*

$$\frac{PT_q}{PT_{opt}} \leq (3 - \frac{2}{q})(1 + \delta).$$

Proof. First we examine the Gantt chart of the schedule from time 0 to PT_q, identifying the total computation and I/O cost, and the idle time. Define the total computation cost as $\pi = \sum_{P_i \in D} f(P_i, P_i) + \sum_{e_{j,i} \in D} f(P_i, P_j)$, where D is the comparison graph generated by two-stage load assignment. Then the total computation and I/O cost is bounded by $\pi(1+\delta)$. Since the scheduling algorithm

assigns a task whenever there is an idle core available, the total idle time in all q cores from time 0 to time PT_q is at most $(q-1)Cost(D)$. Then

$$\max(Cost(D), \frac{\pi}{q}) \leq PT_q \leq \frac{(q-1)Cost(D) + \pi(1+\delta)}{q}.$$

Given an optimal schedule for similarity graph G on q cores, a comparison graph can be derived. Let $Cost_{opt}(G)$ be the largest task cost in this comparison graph. Notice

$$Cost_{min}(G) \leq Cost_{opt}(G).$$

The optimal solution satisfies

$$\max(Cost_{opt}(G), \frac{\pi}{q}) \leq PT_{opt}.$$

Following Theorem 4.3.1,

$$PT_q \leq \frac{q-1}{q} 2(1+\delta)Cost_{min}(G) + (1+\delta)PT_{opt}.$$

Thus

$$\frac{PT_q}{PT_{opt}} \leq \frac{q-1}{q} 2(1+\delta) + (1+\delta) = (3 - \frac{2}{q})(1+\delta).$$

□

Our analysis in the appendix shows that with computation-dominating tasks and a greedy scheduling policy, the upper bound of execution time is affected by the weight of the heaviest task. This supports our load balancing optimization that targets the minimization of the maximum task weight during load assignment.

Stage 1 may produce an unbalanced initial assignment in which some nodes absorb too much computation, especially in dense graphs. Stage 2 mitigates this issue with a sequence of refinements. The following theorem illustrates that for a fully connected graph, our approach delivers a near-optimal solution, and it can be inferred from the proof that the refinement process carried out in Stage 2 is the main reason that this goal is accomplished.

Theorem 4.3.3. *The two-stage load assignment with a greedy scheduler is competitive to the optimum for a fully connected similarity graph with equal partition sizes and equal computation costs in self-comparison and inter-partition comparison. Their relative ratio satisfies*

$$\frac{PT_q}{PT_{opt}} \leq 1 + \delta.$$

Proof. Assume that the number of partitions v is an odd number and we show that all tasks formed have equal weights. The optimality for an even number v can be proved similarly.

Since all nodes have the same self-comparison cost, the same cost to compare with others, and the same cost for communication and data I/O, the cost of each task is proportional to the number of incoming edges for the corresponding node in D. We claim that every node at the end of load assignment has $\frac{v-1}{2}$ incoming edges in comparison graph D, namely it compares with $\frac{v-1}{2}$ neighbors.

We prove by contradiction. If some nodes have the number of incoming edges different from $\frac{v-1}{2}$, then some nodes must have more than $\frac{v-1}{2}$ incoming edges while some other nodes must have less than $\frac{v-1}{2}$ edges since the total number of edges is $\frac{v(v-1)}{2}$ for a fully connected graph. Assume the heaviest nodes P_x has more than $\frac{v-1}{2}$ incoming edges, and there exists an incoming edge from node P_y with the number of incoming edges less than or equals to $\frac{v-1}{2} - 1$. Figure 4.7 illustrates an example with contradiction.

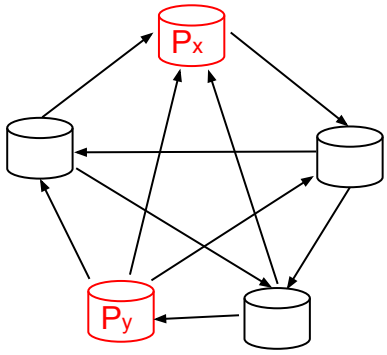


Figure 4.7: An example for proof by contradiction.

Given all partitions have the equal size, Stage 2 of load assignment should not have stopped since it could reverse the edge between T_x and T_y, causing the decrease of $Cost(T_x)$ while $Cost(T_y)$ does not exceed the new value of $Cost(T_x)$. That is a contradiction.

Thus each task T_i formed fetches from its $\frac{v-1}{2}$ neighbors. Tasks have the same weight, leading to a perfect task distribution among q cores. Without loss of

generality, we use $f(P_i, P_i)$, $f(P_i, P_j)$, and $f_c(P_i)$ to represent the cost of self-comparison, inter-partition comparison, and data I/O respectively for all tasks. Then

$$\begin{aligned} PT_q &= \frac{v}{q}(f(P_i, P_i) + f_c(P_i) + \frac{v-1}{2}(f(P_i, P_j) + f_c(P_j))) \\ &\leq \frac{v}{q}(f(P_i, P_i) + \frac{v-1}{2}f(P_i, P_j))(1+\delta). \end{aligned}$$

The above upper bound without factor $1+\delta$ is the lower bound for any schedule including the optimum. Thus the solution derived is within $1+\delta$ of the optimum.

□

4.4 Data Partitioning Optimization

This section presents an improved partitioning method for Phase 1 of partition-based similarity search presented in [5]. The goal of this improvement is twofold: 1) to detect more dissimilarity among partitions to avoid unnecessary data I/O and comparison, and 2) to reduce the size gap among partitions and facilitate the load balancing process.

4.4.1 Dissimilarity Detection with Hölder's Inequality

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 $Sim(d_i, d_j) \leq \|d_i\|_1 \|d_j\|_\infty$, which is a special case introduced in [5].

If the similarity upper-bound is less than τ, such vectors are not similar and comparison between them can be avoided. The algorithm that produces partitions following Hölder's inequality is described as follows.

1. Divide all vectors evenly to produce l consecutive layers $L_1, L_2, \cdots, L_l$ such that all vectors in L_k have lower r-norm values than the ones in L_{k+1}.

2. Subdivide each layer further as follows. For the i-th layer L_i, divide its vectors into i disjoint sub-layers $L_{i,1}, L_{i,2}, \cdots, L_{i,j}$. With $j < i$, members in sub-layer $L_{i,j}$ are extracted from L_i by comparing with the maximum r-norm value in layer L_j:

$$L_{i,j} = \{d_x | d_x \in L_i \text{ and } \max_{d_y \in L_j} \|d_y\|_r < \frac{\tau}{\|d_x\|_s}\}.$$

This partitioning algorithm has a complexity of $O(n \log n)$ for n vectors and can be easily parallelized. Each sub-layer is considered as a data partition and these partitions have dissimilarity relationship with the following property.

Proposition 4.4.1. *Given $i > j$, vectors in sub-layer $L_{i,j}$ are not similar to the ones in any sub-layer $L_{k,h}$ where $k \leq j$ and $k \geq h$.*

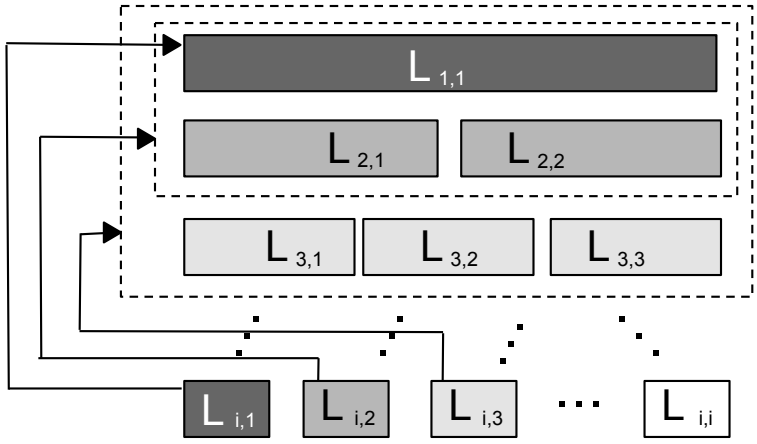


Figure 4.8: Dissimilarity relationship among data partitions.

Figure 4.8 illustrates the dissimilarity relationship among these sub-layers as partitions and each pointing edge represents a dissimilarity relationship. For example, $L_{i,2}$ is not similar to $L_{1,1}, L_{2,1}$, or $L_{2,2}$ in the top two layers.

4.4.2 Even Partition Sizes

To facilitate load balancing in the later phase, we aim at creating more evenly-sized partitions at the dissimilarity detection phase. One way is to divide the large

sub-layers into smaller partitions. Its weakness is that it introduces more potential similarity edges among these partitions, hence the similarity graph produced becomes denser, more communication and I/O overhead are incurred during runtime. Another method targets at approximately the same $L_{i,j}$ size for any $i \leq j$ using a non-uniform layer size. For example, let the size of layer L_k be proportional to the index value k, following the fact that the number of sub-layers in L_k is k in our algorithm. The main weakness of this approach is that less dissimilarity relationships are detected as the top layers become much smaller.

We adopt a hierarchical partitioning that identifies large sub-layers, detects dissimilar vectors inside these sub-layers, and recursively divides them using the procedure discussed in Section 4.4.1. The recursion stops for a sub-layer when reaching a partition size threshold. Each partition inherits the dissimilar relationship from its original sub-layer. The new partitions together with the undivided sub-layers form the undirected similarity graph G ready for load assignment.

4.5 Evaluations

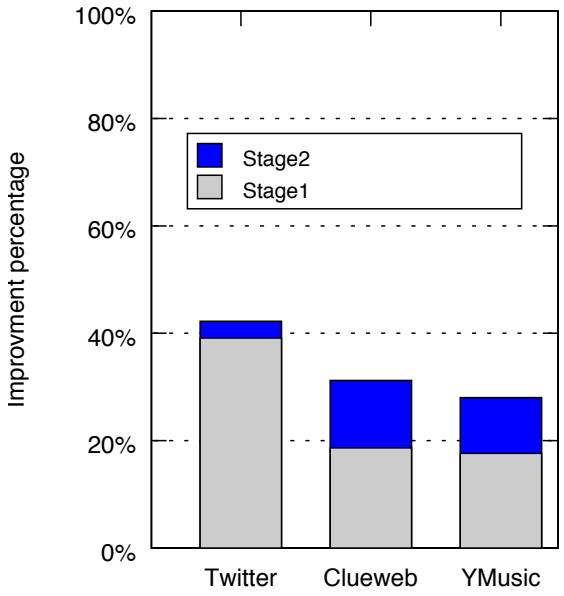
4.5.1 Implementation Details

We have implemented our algorithms in Java using Hadoop MapReduce. Prior to the comparison computation, records are grouped into dissimilar partitions

and this partitioning step including norm value sorting is parallelized. The cost of parallel partitioning is relatively small and is roughly 3% of the total parallel execution time in our experiments. During the load balancing step, the two-stage algorithm defines the comparison direction among potentially similar partitions, generates a comparison graph stored in a distributed cache provided by Hadoop, and derives a set of parallel tasks defined in Algorithm 1.

Hadoop runtime scheduler monitors the load of live nodes in the cluster and assigns a PSS task to the first idle core. Such a dynamic and greedy scheme can absorb potential skewness in data that fluctuates the actual computational cost. Theorem 4.3.2 reflects the competitiveness of PSS tasks scheduled under Hadoop greedy policy. During execution, each task loads the assigned partition with a user-defined reader, obtains a list of partitions to be compared with from the comparison graph file, and loops through the partition list to conduct partition-wise comparison.

In this section, we assess the algorithms using 100 AMD cores for 20M Twitter, 300 cores for 8M ClueWeb, and 20 cores for YMusic. We choose these sizes for faster experimentation while the performance impact of optimization for larger sizes is similar.



(a)

	Circular	2-stage
Max/Avg		
Twitter	2.14	1.45
ClueWeb	4.25	3.25
YMusic	8.97	6.68
Std. Dev/Avg		
Twitter	1.44	0.83
ClueWeb	2.68	1.77
YMusic	2.87	2.06

(b)

Figure 4.9: (a) Parallel time reduction contributed by Stages 1 and 2 compared to the circular assignment. (b) Maximum task cost and standard deviation over the average task cost with circular assignment or with two-stage assignment.

4.5.2 Effectiveness of Two-Stage Load Balance

Figure 4.9(a) shows the improvement percentage in parallel time using two-stage load assignment compared to the baseline circular assignment. Parallel time with two-stage assignment is about 23.2 hours for Twitter, 14 hours for ClueWeb, and 1.7 hours for YMusic respectively. The figure also marks the improvement percentage contributed by Stage 1 and Stage 2 respectively. The overall improvement from the two-stage load assignment is 41% for Twitter, 32% for ClueWeb, and 27% for YMusic. Stage 1 contributes a large portion of the total improvement. Stage 2 contributes about 4% for Twitter, 12% in ClueWeb, and 10% for YMusic. Similarity graphs of ClueWeb and YMusic are denser and Stage 1 can be too aggressive in making the light partitions absorb too much comparison computation. Hence, the refinements in Stage 2 become more effective in such cases.

To examine the weight difference across all tasks, Figure 4.9(b) shows the maximal task weight with circular mapping or with the two-stage balancing method divided by the average task cost. It also lists the cost standard deviation divided by the average task cost. The larger these two ratios are, the more severe load imbalance is. Compared to circular mapping, the two-stage assignment reduces the Max./Avg. ratio by 32.2%, 23.5%, and 25.5% for Twitter, ClueWeb, and YMusic datasets respectively. For Std. Dev./Avg. ratio, the reduction is 42.4%, 34.0%, and 28.2% respectively.

4.5.3 Improved Data Partitioning

Evaluate the performance of the generalized static partitioning algorithm in detecting dissimilarity and narrowing the size gaps among partitions. Figure 4.10 provides a comparison of the improved data partitioning with different r-norms. Y axis is the percentage of pairs detected as dissimilar. r=1 reflects the results in [5]. For ClueWeb, 19% of the total pairs under comparison are detected as dissimilar with r=3 while only 10% for r=1. For Twitter, the percentage of pairs detected as dissimilar is 34% for r=4 compared to 17% for r=1. The results show that choosing r as 3 or 4 is most effective. We have used the best r value for partitioning each dataset.

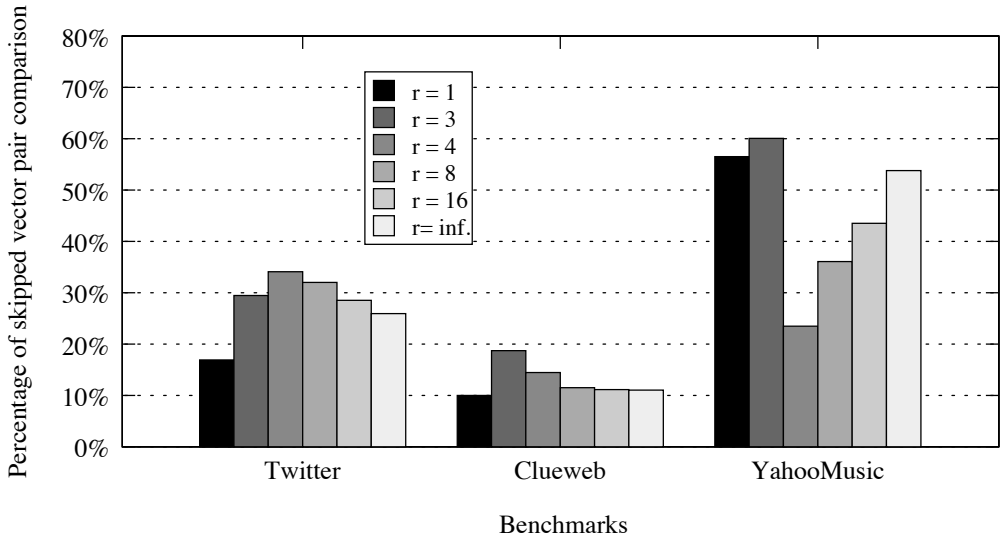


Figure 4.10: Improved partitioning with different r-norms.

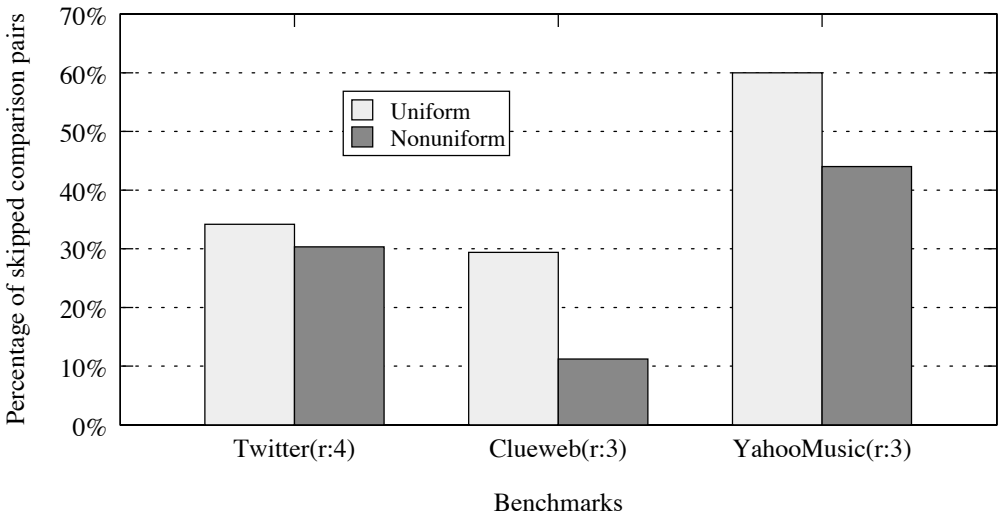


Figure 4.11: Uniform v.s. non-uniform layer size.

As discussed in Section 4.4.2, the initial layer size selection affects the size variation of the final partitions. Figure 4.11 gives a comparison of using uniform layer size and using non-uniform size with the marked r-norm settings. The uniform-sized layers yields better results. For ClueWeb, the uniform layers detect 2.6x as many dissimilar pairs compared to the non-uniform layers. Thus we opt for the uniform layers and recursively apply hierarchical partitioning to even out the sizes of sub-layers.

Table 4.2 shows the effectiveness of recursive hierarchical data partitioning. The ratio of standard deviation of partition sizes over the average size drops by 9.7% for Twitter, 22.3% for ClueWeb, and 3.7% for YMusic. The relatively even

workload benefits the task load balancing process and reduces parallel execution time by 5% to 18% additionally.

Dataset	Std. Dev/Avg (Without)	Std. Dev/Avg (With)	Parallel time reduction
Twitter	1.75	1.58	8.23%
ClueWeb	0.67	0.52	18.23%
YMusic	0.82	0.79	5.29%

Table 4.2: Change of partition sizes and parallel time with or without the recursive hierarchical partitioning.

Chapter 5

Efficient Search Result Ranking in Runtime

5.1 Runtime Search Result Ranking Problem

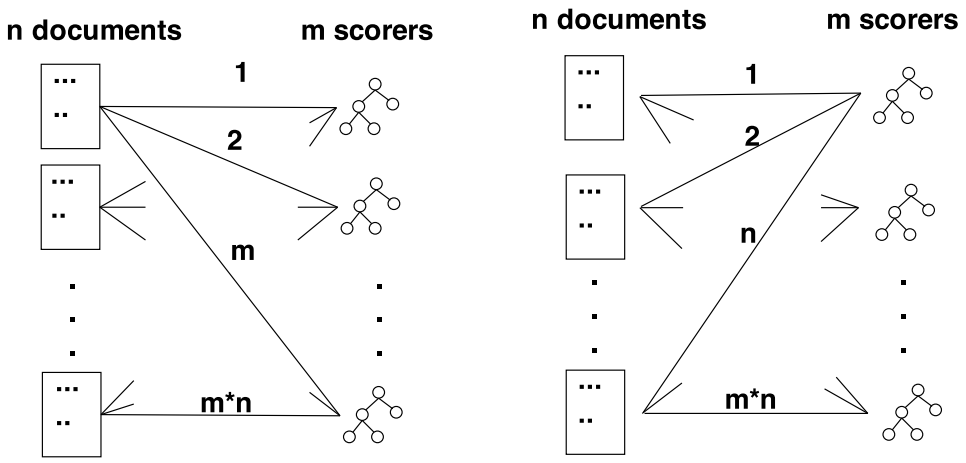
Given a query, there are n documents matching this query and the ensemble model contains m trees. Each tree is called a scorer and contributes a sub-score to the overall score for a document. Following the notation in [19], Algorithm 7 shows the program of DOT. At each loop iteration i, all tress are calculated to gather sub-scores for a document before moving to another document. In implementation, each document is represented as a feature vector and each tree can be stored in a compact array-based format [8]. The time and space cost of updating the

overall score with a sub-score is relatively insignificant. The dominating cost is slow memory accesses during tree traversal based on document feature values. By exchanging loops i and j in Algorithm 7, DOT becomes SOT. Their key difference is the traversal order.

Algorithm 7 Ranking score calculation with DOT.

```
1: for i = 1 to n do
2:     for j = 1 to m do
3:         Compute a sub-score for document i with tree j.
4:         Update document score with the above sub-score.
5:     end for
6: end for
```

Figure 5.1(a) shows the data access sequence in DOT, marked on edges between documents and tree-based scorers. These edges represent data interaction during ranking score calculation. DOT first accesses a document and the first tree (marked as Step 1); it then visits the same document and the second tree. All m trees are traversed before accessing the next document. As m becomes large, the capacity constraint of CPU cache such as L1, L2, or even L3 does not allow all m trees to be kept in the cache before the next document is accessed. The temporal locality of a document is exploited in DOT since the cached copy can be re-accessed many times before being flushed; however, there is no or minimal



(a) (b)

Figure 5.1: Data access order in DOT (a) and SOT (b).

temporal locality exploited for trees. Similarly, Figure 5.1(b) marks data interaction edges and their access order in SOT. SOT traverses all documents for a tree before accessing the next tree. Temporal locality of a tree is exploited in SOT; however, there is no or minimal temporal locality exploited for documents when n is large.

VPred [8] converts if-then-else branches to dynamic data accesses by unrolling the tree depth loop. The execution still follows DOT order, but it overlaps the score computation of several documents to mask memory latency. Such vectorization technique also increases the chance of these documents staying in a cache when processing the next tree. However, it has not fully exploited cache capacity for better temporal locality. Another weakness is that the length of the unrolled

code is quadratic to the maximum tree depth in a ensemble, and linear to the vectorization degree v. For example, the header file with maximum tree depth 51 and vectorization degree 16 requires 22,651 lines of code. Long code causes inconvenience in debugging and code extension. In comparison, our 2D blocking code has a header file of 159 lines.

5.2 2D Block Algorithm

Algorithm 8 is a 2D blocking approach that partitions the program in Algorithm 7 into four nested loops. The loop structure is named SDSD because the first (outer-most) and third levels iterate on tree-based Scorers while the second and fourth levels iterate on Documents. The inner two loops process d documents with s trees to compute sub-scores of these documents. We choose d and s values so that these d documents and s trees can be placed in the fast cache under its capacity constraint. To simplify the presentation, we assume $\frac{m}{s}$ and $\frac{n}{d}$ are integers. The hierarchical data access pattern is illustrated in Figure 5.2. The edges in the left portion of this figure represent the interaction among blocks of documents and blocks of trees with access sequence marked on edges. For each block-level edge, we demonstrate the data interaction inside blocks in the right portion of this figure. Note that there are other variations of 2D blocking structures: SDDS,

DSDS and DSSD. Our evaluation finds that SDSD is the fastest for the tested benchmarks.

Algorithm 8 2D blocking with SDSD structure.

```
1: Instantiate score[] to be zero.
2: for j = 0 to m/s − 1 do
3:     for i = 0 to n/d − 1 do
4:         for jj = 1 to s do
5:             for ii = 1 to d do
6:                 Compute sub-score for document i × d + ii with tree.
7:                 j × s + jj.
8:                 Update the score of this document.
9:             end for
10:        end for
11:    end for
12: end for
```

There are two to three levels of cache in modern AMD or Intel CPUs. For the tested datasets, L1 cache is typically too small to fit multiple trees and multiple document vectors for exploiting temporal locality. Thus L1 is used naturally for spatial locality and more attention is on L2 and L3 cache. 2D blocking design

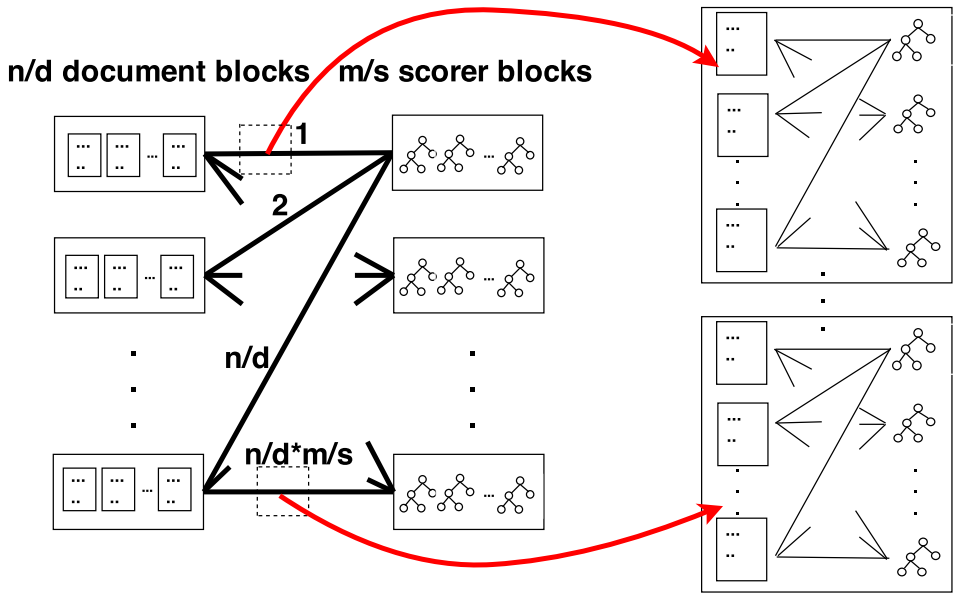


Figure 5.2: Data access order in the SDSD blocking scheme.

allows the selection of s and d values so that s trees and d documents fit in L2 cache.

Detailed cache performance analysis requires a study of cache miss ratio estimation in multiple levels of cache. Here we use a simplified cache-memory model to illustrate the benefits of the 2D blocking scheme. This model assumes there is one level of cache which can hold d document vectors and s tree-based scorers, i.e. space usage for s and d do not exceed cache capacity. Here we estimate the total slow memory accesses during score calculation using the *big O* notation. The inner-most loop ii in Algorithm 8 loads 1 tree and d document vectors. Then loop jj loads another tree and still accesses the same d document vectors. Thus there are a total of $O(s) + O(d)$ slow memory accesses for loops jj and ii. In loop level i, the s trees stay in the cache and every document block causes slow memory accesses, so memory access overhead is $O(s) + O(d) \times \frac{n}{d}$. Now looking at the the outer-most loop j, total memory access overhead per query is $\frac{m}{s}(O(s) + O(n))$ $= O(m + \frac{m \times n}{s})$.

From Figure 5.1, memory access overhead per query in DOT can be estimated as $O(m \times n + n)$ while it is $O(m \times n + m)$ for SOT. Since term $m \times n$ typically dominates, our 2D blocking algorithm incurs s times less overhead in loading data from slow memory to cache when compared with DOT or SOT.

Vectorization in VPred can be viewed as blocking a number of documents and the authors have reported [8] that a larger vectorization degree does not improve latency masking and for Yahoo! dataset, 16 or more degree performs about the same. The objective of 2D blocking scheme is to fully exploit cache locality. We can apply 2D blocking on top of VPred to exploit more cache locality while inheriting the advantages of VPred. We call this approach Block-VPred. The code length of Block-VPred is about the same as VPred.

5.3 Evaluations

2D block and Block-VPred methods are implemented in C and VPred code is from [8]. Code is compiled with GCC using optimization flag -O3. Experiments are conducted on a Linux server with 8 cores of 3.1GHz AMD Bulldozer FX8120 and 16GB memory. FX8120 has 16KB of L1 data cache per core, 2MB of L2 cache shared by two cores, 8MB of L3 cache shared by eight cores. The cache line is of size 64 bytes. Experiments are also conducted in Intel X5650 2.66GHz six-core dual processors and the conclusions are similar. The following we report results from AMD processors.

We use the following learning-to-rank datasets as the core test benchmarks. (1) Yahoo! dataset [20] with 709,877 documents and 519 features per document from its learning-to-rank challenge. (2) MSLR-30K dataset [2] with 3,771,125

documents and 136 features per document. (3) MQ2007 dataset [1] with 69,623 documents and 46 features per document. The tree ensembles are derived by the open-source jforests [28] package using LambdaMART [18]. To assess score computation in presence of a large number of trees, we have also used bagging methods to combine multiple ensembles and each ensemble contains additive boosting trees.

There are 23 to 120 documents per query labeled in these datasets. In practice, a search system with a large dataset ranks thousands or tens of thousands of top results after the preliminary selection. We synthetically generate more matched document vectors for each query. Among these synthetic vectors, we generate more vectors bear similarity to those with low labeled relevance scores, because typically the majority of matched results are less relevant.

Metrics. We mainly report the average time of computing a sub-score for each matched document under one tree. This scoring time multiplied by n and m is the scoring latency per query for n matched documents ranked with an m-tree model. Each query is executed by a single core.

5.3.1 Scoring Time

Table 5.1 lists scoring time under different settings. Column 2 is the maximum number of leaves per tree. Tuple [s,d,v] includes the parameters of 2D blocking and the vectorization degree of VPred that leads to the fastest scoring time. Choices of

Dataset	Leaves	m	n	DOT	SOT	VPred [v]	2D blocking [s, d]	Block-VPred [s, d, v]	Latency
Yahoo!	50	7,870	5,000	186.0	113.8	47.4 [8]	36.4 [300, 300]	36.7 [300, 320, 8]	1.43
	150	8,051	2,000	377.8	150.2	123.0 [8]	81.9 [100, 400]	76.1 [100, 480, 8]	1.23
	400	2,898	5,000	312.3	223.8	136.2 [8]	90.9 [100, 400]	86.0 [100, 400, 8]	1.25
MSLR-30K	50	1,647	5,000	88.3	41.4	32.6 [8]	26.6 [500, 1,000]	31.1 [500, 1,600, 8]	0.22
MQ2007	50	9,870	10,000	1.79	1.66	2.02 [8]	1.51 [300, 5,000]	1.94 [300, 5,000, 8]	0.15
	200	10,103	10,000	204.1	30.3	43.1 [32]	28.3 [100, 10,000]	26.2 [100, 5,000, 32]	2.65

Table 5.1: Scoring time per document per tree in nanoseconds for five algorithms. Last column shows the average scoring latency per query in seconds under the fastest algorithm marked in gray.

v for VPred are the best in the tested AMD architecture and are slightly different from the values reported in [8] with Intel processors. Last column is the average scoring latency per query in seconds after visiting all trees. For example, 2D blocking is 361% faster than DOT and is 50% faster than VPred for Row 3 with Yahoo! 150-leaf 8,051-tree benchmark. In this case, Block-VPred is 62% faster than VPred and each query takes 1.23 seconds to complete scoring with Block-VPred. For a smaller tree in Row 5 (MSLR-30K), Block-VPred is 17% slower than regular 2D blocking. In such cases, the benefit of converting control dependence as data dependence does not outweigh the overhead introduced.

Figure 5.3 shows the scoring time for Yahoo! dataset under different settings. In Figure 5.3(a), n is fixed as 2,000; DOT time rises dramatically when m in-

creases because these trees do not fit in cache; SOT time keeps relatively flat as m increases. In Figure 5.3(b), m is fixed as 8,051 while n varies from 10 to 100,000. SOT time rises as n grows and 2D blocking is up to 245% faster. DOT time is relatively stable. 2D blocking time and its gap to VPred are barely affected by the change of m or n. Block-VPred is 90% faster than VPred when n=5,000, and 100% faster when n=100,000. Figure 5.3(c) shows the 2D blocking time when varying s and d. The lowest value is achieved with s=1,000 and d=100 when these trees and documents fit in L2 cache.

5.3.2 Cache Behavior

Linux *perf* tool reports L1 and L3 cache miss ratios during execution. We observed no strong correlation between L1 miss ratio and scoring time. L1 cache allows program to exploit limited spatial locality, but is too small to exploit temporal locality in our problem context. L3 miss ratio does show a strong correlation with scoring time. In our design, 2D blocking sizes (s and d) are determined based on L2 cache size. Since L2 cache is about the same size as L3 per core in the tested AMD machine, reported L3 miss ratio reflects the characteristics of L2 miss ratio.

Figure 5.4 plots the L3 miss ratio under the same settings as Figure 5.3 for Yahoo! data. This ratio denotes among all the references to L3 cache, how many are missed and need to be fetched from memory. The ratios of Block-VPred,

which are not listed, are very close to that of 2D blocking. In Figure 5.4(a) with n=2,000, SOT has a visibly higher miss ratio because it needs to bring back most of the documents from memory to L3 cache every time it evaluates them against a scorer; n is too big to fit all documents in cache. The miss ratio of DOT is low when all trees can be kept in L2 and L3 cache; this ratio grows dramatically after m=500. Figure 5.4(b) shows miss ratios when m=8,051 and n varies. The miss ratio of SOT is close to VPred and 2D blocking when n<100, but deteriorates significantly when n increases and these documents cannot fit in cache any more. The miss ratios of VPred in both Figure 5.4(a) and 5.4(b) are below 6% because vectorization improves cache hit ratio. Performance of 2D blocking is the best, maintaining miss ratio around 1% even when m or n is large.

Figure 5.4(c) plots L3 miss ratio of 2D blocking when varying s and d block sizes. The trends are strongly correlated with the scoring time curve in Figure 5.3(c). The optimal point is reached with s=1,000 and d=100 when these trees and documents fit in L2 cache. When s=1,000, miss ratio varies from 1.64% (d=100) to 78.1% (d=100,000). As a result, scoring time increases from 86.2ns to 281.5ns.

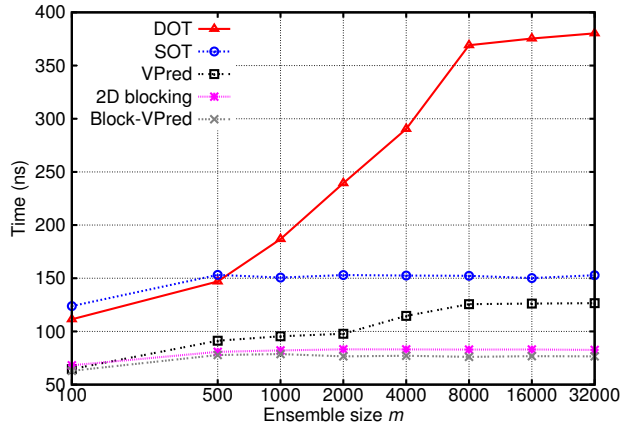
5.3.3 Branch Mis-prediction Rate

We have also collected instruction branch mis-prediction ratios during computation. For MQ2007 and 50-leaf trees, mis-prediction ratios of DOT, SOT, VPred, 2D blocking and Block-VPred are 1.9%, 3.0%, 1.1%, 2.9%, and 0.9% respectively. For 200-leaf trees, these ratios increase to 6.5%, 4.2%, 1.2%, 9.0%, and 1.1%. VPred's mis-prediction ratio is lower than 2D blocking while its scoring time is still longer, indicating the impact of cache locality on scoring time is bigger than branch mis-prediction. For smaller trees, mis-prediction ratios of 2D blocking and Block-VPred are close and this explains why Block-VPred does not outperform 2D blocking in Table 5.1 for 50-leaf trees. Adopting VPred's strategy of converting if-then-else instructions pays off for large trees. For such cases when n increases, Block-VPred outperforms 2D blocking with lower branch mis-prediction ratios. This is reflected in the Yahoo! 150-leaf 8,051-tree benchmark: mis-prediction ratios are 1.9%, 2.7%, 4.3%, and 6.1% for 2D blocking, 1.1%, 0.9%, 0.84%, and 0.44% for Block-VPred, corresponding to the cases of n=1,000, 5,000, 10,000 and 100,000 respectively.

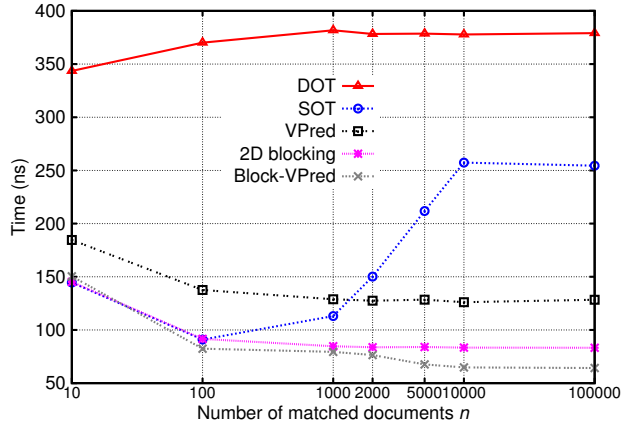
5.3.4 Parallelism & Combined Processing

Multi-tree score calculation of each query can be conducted in parallel on multiple cores to further reduce latency. Our experiments show that 2D blocking still

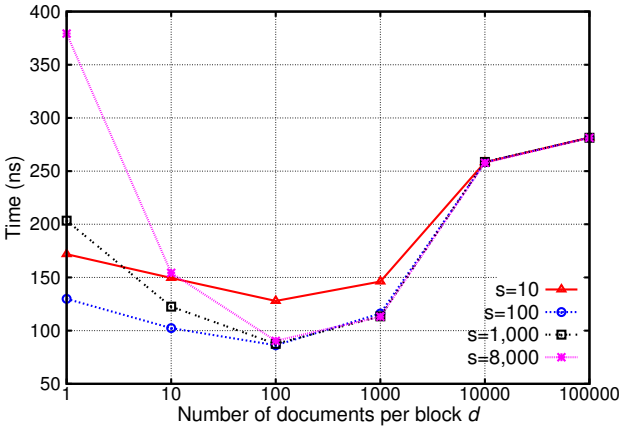
maintains its advantage using multiple threads. In some applications, the number of top results (n) for each query is inherently small and can be much smaller than the optimal block size (d). In such cases, multiple queries could be combined and processed together to fully exploit cache capacity. Our experiments with Yahoo! dataset and 150-leaf 8,051-tree ensemble shows that combined processing could reduce scoring time per query by 12.0% when n=100, and by 48.7% when n=10.



(a)

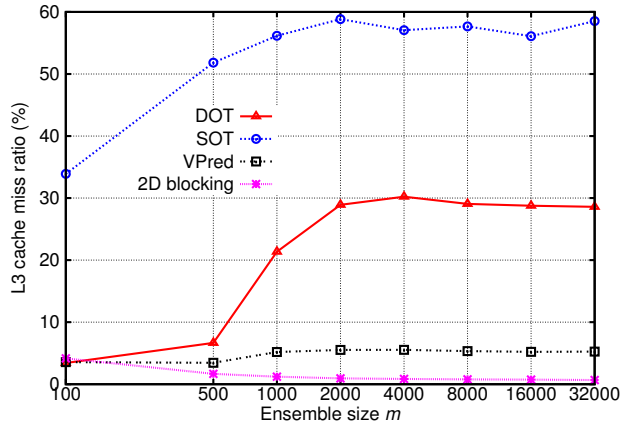


(b)

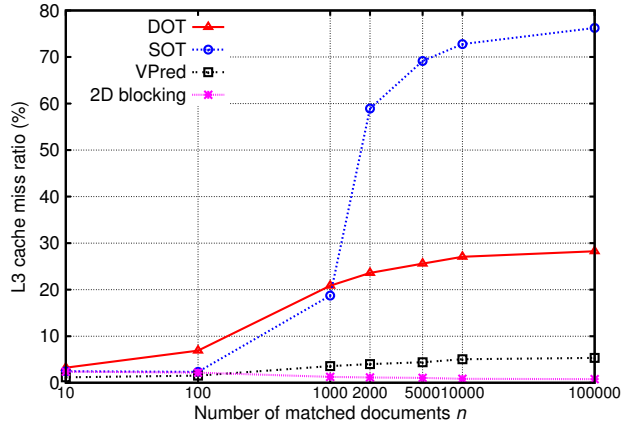


(c)

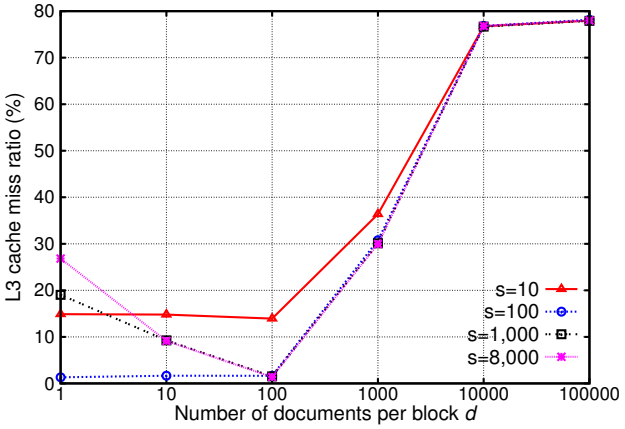
Figure 5.3: Scoring time per document per tree in nanoseconds when varying m (a) and n (b) for five algorithms, and varying s and d for 2D blocking (c). Benchmark used is Yahoo! dataset with a 150-leaf multi-tree ensemble.



(a)



(b)



(c)

Figure 5.4: L3 miss ratio when varying n (a), varying m (b) for four algorithms, and when varying s and d for 2D blocking (c).

Chapter 6

Conclusions and Future Work

The contribution of this dissertation work could be summarized as three parts.

- **Cache-conscious partition-based similarity search.** We propose and develop a partitioned similarity search algorithm with cache-conscious data layout and traversal. 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 such that the size of temporary vectors accessed can be controlled and fit in the fast cache; 2)

coalesce vectors with size-controlled inverted indexing such that the temporal locality of data elements visited can be exploited. Our analysis provides a guidance for optimal parameter setting. The evaluation result shows that the optimized code can be upto 2.74x as fast as the original cache-oblivious design. Vector coalescing is more effective if there is a decent number of features shared among the coalesced vectors. We also discuss how to further accelerate similarity search by incorporating incremental computing and approximation methods such as Locality Sensitive Hashing. Introducing LSH step makes PSS one to two orders of magnitude faster with only 3% recall drop.

- **Two-stage load balance.** We propose and implement a two-stage load balancing algorithm for efficiently executing partition-based similarity search in parallel. The first stage constructs a preliminary load assignment over tasks. The second stage refines the assignment for denser graphs. The analysis provided shows its competitiveness to the optimal solution with constant ratios, for both task load balancing and parallel runtime. We also present an improved and hierarchical static data partitioning method to detect dissimilarity and even out the partitions sizes. Our experiments demonstrate that the two-stage load assignment improves the circular assignment by up to 41% in the tested datasets. The improved static partitioning avoids more

unnecessary I/O and communication and reduces the size gaps among partitions with up to 18% end-performance gain in the tested cases.

- **Search result ranking in runtime.** We propose a cache-conscious design for computing ranking scores with a large number of trees and/or documents by exploiting memory hierarchy capacity for better temporal locality. While ranking accuracy is maintained to be the same, our experiments show that 2D blocking can be up to 620% faster than DOT, up to 214% faster than SOT, and 54% faster than VPred. Apply 2D blocking on the top of VPred which has advantages in reducing branch mis-prediction, the blocked code is up to 76% faster than VPred.

There are various aspects in APSS that are worthy of further study. The impact of a multi-user computing cluster environment on parallel similarity search algorithms is an interesting topic to explore. We can also study how the runtime computing resource per thread changes when more threads are running concurrently and how the number of CPU cores per machine affects the algorithm design. Our 2D blocking technique is studied in the context of tree-based ranking ensembles and one of future work is to extend it for other types of ensembles by iteratively selecting a fixed number of the base rank models that fit the fast cache.

Bibliography

[1] Lector 4.0 datasets. http://research.microsoft.com/en-us/um/beijing/projects/letor/letor4dataset.aspx.

[2] Microsoft learning to rank datasets. http://research.microsoft.com/en-us/projects/mslr/.

[3] Fabio Aiolli. Efficient top-n recommendation for very large scale binary rated datasets. In *Proc. of 7th ACM Conf. on Recommender Systems*, pages 273–280, 2013.

[4] Maha Alabduljalil, Xun Tang, and Tao Yang. Cache-conscious performance optimization for similarity search. In *ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 713–722, 2013.

[5] Maha Alabduljalil, Xun Tang, and Tao Yang. Optimizing parallel algorithms for all pairs similarity search. In *ACM Inter. Conf. on Web Search and Data Mining (WSDM)*, pages 203–212, 2013.

[6] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *Proceedings of the 32nd international conference on Very large Data Bases (VLDB)*, 2006.

[7] Nima Asadi and Jimmy Lin. Training Efficient Tree-Based Models for Document Ranking. In *the 35th European Conference on Information Retrieval (ECIR)*, pages 146–157, 2013.

[8] Nima Asadi, Jimmy Lin, and Arjen P De Vries. Runtime Optimizations for Tree-Based Machine Learning Models. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, pages 1–13, 2013.

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

[10] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[11] C.S. Badue, R. Baeza-Yates, B. Ribeiro-Neto, a. Ziviani, and N. Ziviani. Analyzing imbalance among homogeneous index servers in a web search system. *Information Processing & Management*, 43(3):592–608, May 2007.

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

[13] Ranieri Baraglia, Gianmarco De Francisci Morales, and Claudio Lucchese. Document similarity self-join with mapreduce. In *2010 IEEE International Conference on Data Mining (ICDM)*, pages 731–736.

[14] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *Proc. of Inter. Conf. on World Wide Web*, WWW '2007, pages 131–140. ACM.

[15] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Journal of Parallel and Distributed Computing*, pages 207–216, 1995.

[16] Peter Boncz, Data Distilleries B V, Stefan Manegold, and Martin L Kersten. Database Architecture Optimized for the new Bottleneck: Memory Access. In *Proceedings of the 25th international conference on Very large Data Bases (VLDB)*, 1999.

[17] A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, pages 21–, Washington, DC, USA, 1997. IEEE Computer Society.

[18] Christopher J. C. Burges, Krysta Marie Svore, Paul N. Bennett, Andrzej Pastusiak, and Qiang Wu. Learning to rank using an ensemble of lambda-gradient models. In *J. of Machine Learning Research*, pages 25–35, 2011.

[19] B Barla Cambazoglu, Hugo Zaragoza, Olivier Chapelle, and Jiang Chen. Early Exit Optimizations for Additive Machine Learned Ranking Systems Ranking in Additive Ensembles. In *Proceedings of the 6th ACM International Conference on Web Search and Data Mining (WSDM)*, pages 411–420, 2010.

[20] Olivier Chapelle and Yi Chang. Yahoo! Learning to Rank Challenge Overview. *J. of Machine Learning Research*, pages 1–24, 2011.

[21] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 380–388, New York, NY, USA, 2002. ACM.

[22] Abdur Chowdhury, Ophir Frieder, David A. Grossman, and M. Catherine McCabe. Collection statistics for fast duplicate document detection. *ACM Transactions of Information Systems*, 20(2):171–191, 2002.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on*

Opearting Systems Design & Implementation - Volume 6, OSDI'04, pages 137–150.

[24] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990.

[25] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Trans. Math. Softw.*, 28(2):239–267, June 2002.

[26] Tamer Elsayed, Jimmy Lin, and Douglas W. Oard. Pairwise document similarity in large collections with mapreduce. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, HLT-Short '08, pages 265–268.

[27] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[28] Yasser Ganjisaffar, Rich Caruana, and Cristina Lopes. Bagging Gradient-Boosted Trees for High Precision, Low Variance Ranking Models. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 85–94, 2011.

[29] M. R. Garey and R. L. Grahams. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4:187–200, 1975.

[30] Pierre Geurts and Gilles Louppe. Learning to rank with extremely randomized trees. *J. of Machine Learning Research*, 14:49–61, 2011.

[31] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[32] Andrey Gulin, Igor Kuralenok, and Dmitry Pavlov. Winning the transfer learning track of yahoo!'s learning to rank challenge with yetirank. *J. of Machine Learning Research*, 14:63–76, 2011.

[33] Hannaneh Hajishirzi, Wen-tau Yih, and Aleksander Kolcz. 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, pages 419–426, New York, NY, USA, 2010. ACM.

[34] Heimsund Halliday. http://code.google.com/p/matrix-toolkits-java.

[35] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, 1998.

[36] Luo Jie, Sudarshan Lamkhede, Rochit Sapra, Evans Hsu, Helen Song, and Yi Chang. A unified search federation system based on online user feedback. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1195–1203, New York, NY, USA, 2013. ACM.

[37] Nitin Jindal and Bing Liu. Opinion spam and analysis. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, WSDM '08, pages 219–230.

[38] David Kanter. Md's bulldozer microarchitecture. *realworldtech.com*, 2010.

[39] Enver Kayaaslan, Simon Jonassen, and Cevdet Aykanat. A Term-Based Inverted Index Partitioning Model. *ACM Transactions on the Web (TWEB)*, 7(3):1–23, 2013.

[40] David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel*, 2009.

[41] Jimmy Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *Proceedings of the 32Nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, pages 155–162.

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

[43] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Detectives: detecting coalition hit inflation attacks in advertising networks streams. In *Proc. of WWW'2007*, pages 241–250. ACM.

[44] Gianmarco De Francisci Morales, Claudio Lucchese, and Ranieri Baraglia. Scaling out all pairs similarity search with mapreduce. In *8th Workshop on Large-Scale Distri. Syst. for Information Retrieval*, 2010.

[45] Dmitry Yurievich Pavlov, Alexey Gorodilov, and Cliff A. Brunk. Bagboo: a scalable hybrid bagging-the-boosting model. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, pages 1897–1900, 2010.

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

[47] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th Inter-*

national Conference on Very Large Data Bases, VLDB '94, pages 510–521. Morgan Kaufmann Publishers Inc, 1994.

[48] Kai Shen, Tao Yang, and Xiangmin Jiao. S+: Efficient 2d sparse lu factorization on parallel machines. *SIAM J. Matrix Anal. Appl.*, 22(1):282–305, April 2000.

[49] Narayanan Shivakumar and Hector Garcia-Molina. Building a scalable and accurate copy detection mechanism. In *Proceedings of the First ACM International Conference on Digital Libraries*, DL '96, pages 160–168.

[50] Martin Theobald, Jonathan Siddharth, and Andreas Paepcke. Spotsigs: robust and efficient near duplicate detection in large web collections. In *SIGIR '08: Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 563–570, 2008.

[51] Ferhan Ture, Tamer Elsayed, and Jimmy Lin. 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, pages 943–952, 2011.

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

[53] Richard Vuduc, James W. Demmel, Katherine A. Yelick, Shoaib Kamil, Rajesh Nishtala, and Benjamin Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–35, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[54] Ye Wang, Ahmed Metwally, and Srinivasan Parthasarathy. Scalable all-pairs similarity search in metric spaces. *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 829–837, 2013.

[55] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *Proceeding of the 17th international conference on World Wide Web*, pages 131–140. ACM, 2008.

[56] Matei Zaharia, Khaled Elmeleegy, Dhruba Borthakur, Scott Shenker, Joydeep Sen Sarma, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. of EuroSys*, pages 265–278, 2010.