L2AP: Fast Cosine Similarity Search With Prefix L-2 Norm Bounds

David C. Anastasiu and George Karypis
Department of Computer Science and Engineering
University of Minnesota, Twin Cities, MN 55455, U.S.A.
{dragos, karypis}@cs.umn.edu

***Abstract*—The *All-Pairs similarity search*, or *self-similarity join* problem, finds all pairs of vectors in a high dimensional sparse dataset with a similarity value higher than a given threshold. The problem has been classically solved using a dynamically built *inverted index*. The search time is reduced by early pruning of candidates using size and value-based bounds on the similarity. In the context of cosine similarity and weighted vectors, leveraging the Cauchy-Schwarz inequality, we propose new ℓ^2-norm bounds for reducing the inverted index size, candidate pool size, and the number of full dot-product computations. We tighten previous candidate generation and verification bounds and introduce several new ones to further improve our algorithm's performance. Our new pruning strategies enable significant speedups over baseline approaches, most times outperforming even approximate solutions. We perform an extensive evaluation of our algorithm, `L2AP`, and compare against state-of-the-art exact and approximate methods, `AllPairs`, `MMJoin`, and `BayesLSH`, across a variety of real-world datasets and similarity thresholds.**

I. Introduction

Similarity search is a crucial component in many real-world applications, such as near-duplicate document detection [1], query refinement [2], [3], clustering [4], [5], and collaborative filtering [6]. Given a function sim$(\boldsymbol{x}, \boldsymbol{y})$ and a threshold t, *similarity search* finds all objects in a dataset with a similarity value of at least t when compared to some query object. *All-Pairs similarity search* (APSS) is a more specific version of the problem, where the search is repeated for each object.

Objects in the real-world are generally numerically represented by high dimensional vectors, where each dimension is a feature extracted from the object. When only the presence of features is of interest, binary vectors suffice to encode the set of features in an object. However, weighted vectors often better represent objects for search [7], [8], and are standard in fields like information retrieval [9] and text mining [10].

A naïve approach to solving APSS executes O(n^2) object comparisons for a dataset of n objects. Exact algorithms for solving the problem thus rely on filtering techniques that allow pruning most of the $n - 1$ possible similarity candidates for each query object. They often proceed in two stages. First, during **candidate generation**, a list of objects is compiled whose similarity scores to the query object are believed to exceed the threshold. Potential candidates during this stage are vetted based on their *prefix*, *size*, *suffix* and *length*, which we describe below. The **candidate verification** stage finalizes the similarity computation for identified candidates and compares it against the threshold. Additional pruning based on suffix size or magnitude reduce the number of full similarities being computed.

Prefix filtering allows identifying candidates based only on a small number of prefix features, given a global feature ordering [11]. *Size* and *suffix filtering* eliminate candidates based on the number of their non-zero values (size), or based on suffix similarity estimates [3]. *Length filtering* [8] was recently introduced, which uses the magnitude of the suffix to further reduce prefix length (and thus index size), intending to reduce the size of the candidate set. As we will show, length filtering based index reduction can only be effective for similarity values higher than 0.5. However, many emerging applications (e.g. 3D scene reconstruction, plagiarism detection) require the use of relatively low similarity thresholds [12].

In this paper, we introduce new filtering strategies that allow the exact APSS problem to be solved efficiently for *cosine similarity* of *weighted vectors*. *Prefix filtering* based on the ℓ^2-norm achieves better index reduction than previous approaches. Additionally, we propose *positional* and enhanced *suffix filtering* methods that successfully prune most false positive candidates, resulting in relatively few object pairs having their similarity value computed in full. While previous algorithms do not scale well as the similarity threshold decreases, our new pruning techniques make our method effective at both high and low similarity thresholds.

The main contributions of this paper are as follows:

- We provide more effective *ℓ^2-norm filtering* by leveraging the Cauchy-Schwarz inequality. We show that ℓ^2-norm bounds lead to smaller prefix sizes and allow pruning more candidates than *length filtering* in general.
- We provide tighter *positional* and *suffix*-based bounds that allow pruning the majority of candidates without fully performing the similarity computation. This complements our *ℓ^2-norm filtering* and leads to orders of magnitude improvement over previous approaches.
- Based on these ideas, we develop a new filtering-based exact cosine similarity search algorithm called `L2AP`, which works well at both high and low similarity thresholds. We evaluate our prototype experimentally on a variety of real-world datasets and similarity thresholds.
- We develop `L2AP-approx`, in which we extend `L2AP` with state-of-the-art approximate `BayesLSH-Lite` candidate pruning, and compare against our exact algorithm variant. In most cases, we find that `L2AP`'s filtering strategies out-

perform approximate candidate pruning in L2AP-approx and other baselines.

The remainder of the paper is organized as follows. Section II introduces the problem and notation used throughout the paper. Section III summarizes existing approaches to solving APSS. Section IV details existing filtering techniques, which we extend in our proposed algorithms, L2AP and L2AP-approx, presented in Section V. We describe our evaluation methodology and analyze experimental results in Section VI, and Section VII concludes the paper.

II. Problem Statement

Let a set of real-valued vectors be represented as the rows of a matrix D of size $n \times m$, and let $\text{sim}(\boldsymbol{x}, \boldsymbol{y})$ be a commutative similarity function between row vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ of D. Given a similarity threshold t, we solve the APSS problem by computing the similarity values of all row pairs (x, y) having $\text{sim}(\boldsymbol{x}, \boldsymbol{y}) \geq t$, for all x, y in D, $x \neq y$. Note that x is a row ID, and $\boldsymbol{x}$ (bold) is the row vector associated with row x in D. Assuming $\text{sim}(\cdot, \cdot)$ to be the cosine similarity and that all rows in D have been normalized to have unit length ($||\boldsymbol{x}||_2 = 1, \ \forall\ x$ in D), the similarity computation reduces to finding the dot-product between the two vectors,

$$\text{sim}(\boldsymbol{x}, \boldsymbol{y}) = \text{dot}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}\boldsymbol{y}^T = \sum_{j=1}^{m} x_j \times y_j,$$

where x_j is the j-th feature of $\boldsymbol{x}$, i.e. element $D(x, j)$. Going forward, we will note the *ℓ^2-norm*, *magnitude*, or *length* of a vector by the shorthand $||\boldsymbol{x}|| = ||\boldsymbol{x}||_2 = \sqrt{\sum_{j=1}^{m} x_j{}^2}$.

We denote by $|\boldsymbol{x}|$ the size of, or number of non-zeros in the vector $\boldsymbol{x}$, not to be confused with its length, $||\boldsymbol{x}||$. At times, we are interested in the maximum value of some row i in D, which we denote by rw_i, of some vector $\boldsymbol{x}$, which we denote by rw_x, or of some column j, which we denote by cw_j. Let $\Sigma_x = \sum_{j=1}^{m} x_j$ be the sum of elements in vector $\boldsymbol{x}$.

An *inverted index* representation of D is a set of m lists, $\mathcal{I} = \{I_1, I_2, \ldots, I_m\}$, one for each column in D, where list I_j contains pairs $(x, D(x, j))$ s.t. x is a row in D, and $D(x, j) = x_j \neq 0$. We construct the inverted index dynamically, and choose to index only a subset of the less frequent features, the **suffix** of the vector, which we denote as $\boldsymbol{x}'' = \boldsymbol{x}''_p = \langle 0, \ldots, 0, x_p, \ldots, x_m \rangle$, where p is the first feature we index in the vector. We denote by rw''_x the maximum value in the suffix $\boldsymbol{x}''$, and by Σ''_x the sum of the suffix elements. Similarly, the **prefix** of $\boldsymbol{x}$ is noted as $\boldsymbol{x}' = \boldsymbol{x}'_p = \langle x_1, \ldots, x_{p-1}, 0, \ldots, 0 \rangle$, and has maximum value rw'_x and element sum Σ'_x. Given these representations of prefix and suffix, one can verify that,

$$\begin{aligned}
\boldsymbol{x} &= \boldsymbol{x}' + \boldsymbol{x}'', \\
||\boldsymbol{x}||^2 &= ||\boldsymbol{x}'||^2 + ||\boldsymbol{x}''||^2, \\
\text{dot}(\boldsymbol{x}, \boldsymbol{y}) &= \sum_{j=1}^{p-1} x_j \times y_j + \sum_{j=p}^{m} x_j \times y_j, \\
&= \text{dot}(\boldsymbol{x}', \boldsymbol{y}) + \text{dot}(\boldsymbol{x}'', \boldsymbol{y}).
\end{aligned}$$

III. Related Work

APSS has its roots in the *similarity join* problem from the database community [11], [13]. Chaudhuri et al. first formalized the *prefix-filtering* principle [11], showing that only a few elements from the beginning of a query vector must be checked against other vectors to find all necessary candidates. Bayardo et al. [3] disconnected the problem from the underlying database system and developed additional pruning strategies based on a pre-defined vector order in the dataset. They also introduced *dynamic indexing*, leveraging the prefix-filtering principle to index only a portion of each vector after its candidate list was generated. The majority of subsequently developed methods for solving the APSS problem follow the same format as in Bayardo et al.'s method.

A number of extensions focused on the set-based or binary representation of objects. Xiao et al. [1] first introduced a tighter indexing bound and *positional filtering* in PPJoin. During candidate generation, positional filtering provides additional pruning based on the remaining size of the vectors once a feature is found in common. Xiao et al. then push filtering into the candidate verification stage through Hamming distance based *suffix filtering*. When considering string similarity search using the edit distance measure, Xiao et al. [14] show that the problem can be efficiently solved using q-gram-based mismatch filtering. Xiao et al. [15] provide further AllPairs optimizations to answer top-k queries efficiently.

While previous algorithms focused on reducing the generated candidate pool size, Ribeiro and Härder [16] sought to reduce overall search time by minimizing the size of the inverted index through dynamic min-prefix indexing. They coupled a cheap candidate generation step with additional stoping criteria in the verification stage to improve on AllPairs and PPJoin. Wang et al. [17] sought to reach a balance between the prefix and candidate pool sizes and developed a cost-based scheme for choosing variable-length prefixes. We borrow some ideas from set-based similarity search algorithms, such as positional and suffix filtering, but their details are not applicable, as we focus on cosine similarity search of *weighted* vectors.

There has been little focus, in comparison, on solving the APSS problem for weighted vectors and cosine similarity. Bayardo et al. [3] gave the first integrated solution for the problem, the AllPairs algorithm. In APT, Awekar and Samatova [18] provide tighter bounds over AllPairs on the candidate vector minimum size and similarity score estimate. Lee et al. [8] introduce *length filtering* and length-based *suffix filtering* in MMJoin. As they are pertinent to our problem, we detail these methods in Section IV. We then show, both theoretically and experimentally, that our pruning strategies outperform those in these methods.

Although emphasis has recently shifted to solving APSS exactly, approximate methods remain popular, especially in domains only interested in objects with high similarity thresholds. In the context of near-duplicate object detection, Broder et al. [4] apply similarity search to sketches built using min-wise independent permutations of shingled Web documents.

A popular alternative, Locality Sensitive Hashing (LSH) [19], [20], uses families of functions that hash similar objects to the same bucket with high probability to generate candidate sets. Zhai et al. [12] present a probabilistic algorithm for similarity search based on random filters. Satuluri and Parthasarathy [21] introduce `BayesLSH`, a principled Bayesian approach for candidate pruning and similarity estimation, which they combine with candidate generation steps from `AllPairs` and `LSH`. Unlike approximate methods, our method outputs all objects at or above the similarity threshold. Yet we show that, in most cases, `L2AP` outperforms `BayesLSH`. `L2AP` often performs better even than our approximate variant, `L2AP-approx`, showing that pruning the search space is a very effective strategy for solving the APSS problem.

IV. Filtering Framework

`AllPairs` is an algorithm for exact all-pairs similarity search introduced by Bayardo et al. [3] and extended by many other filtering-based APSS approaches. Our extension, `L2AP`, improves `AllPairs` by obtaining tighter similarity bounds in all stages of the algorithm. Here, we detail the filtering framework in `AllPairs` and subsequent extensions.

One could solve the APSS problem by finding all nearest neighbors in the dataset for each vector. However, given a sparse dataset, a vector $\boldsymbol{x}$ may not have features in common with many candidate vectors. `AllPairs` avoids computing $\boldsymbol{x}$'s similarity with these vectors by using an inverted index, a set of lists, one for each feature, containing vectors with non-zero values in D for that feature, and their associated feature values. One can then traverse the inverted lists for only the terms in $\boldsymbol{x}$ to find its possible neighbors. Score accumulation using the values stored in the index can be used to compute the similarity value, and the original vector can be discarded [22].

`AllPairs` improves these standard similarity search techniques in several ways. It builds the index dynamically and exploits the threshold t and a pre-defined sort order on D to limit the feature values being indexed, the candidate pairs being generated, and for which candidate pairs the exact similarity value should be computed. Algorithms 1 and 2 present the pseudo-code for `AllPairs`. As we continue, we will also detail pruning strategies employed in subsequent extensions `APT` and `MMJoin`.

A. Prefix and Suffix Filtering

Chaudhuri et al. introduced the *prefix-filtering* principle, which has been used to limit the size of the inverted index. It states informally that, given a global ordering of features in the dataset, one can stop indexing features in $\boldsymbol{x}$ as soon as they can ensure that $\boldsymbol{x}$ will have at least one feature in the index in common with all its *true neighbors* (those vectors $\boldsymbol{y}$ s.t. $\text{sim}(\boldsymbol{x}, \boldsymbol{y}) \geq t$). Chaudhuri et al. and Lee et al. order their datasets in increasing column frequency order and index features at the beginning of $\boldsymbol{x}$, i.e. its *prefix*. They use the remaining part of the vector, its *suffix*, to estimate and complete similarity computations. While they do not expressly state it, Bayardo et al. also use the *prefix-filtering* principle in their algorithm, `AllPairs`. Yet they choose an opposite ordering of the features, index the suffix of each vector, and use the prefix to complete the similarity computation. To avoid confusion, we will refer to prefix filtering, henceforward, as *index filtering*, since its goal is to reduce the index size. Similarity, we will refer to suffix filtering as *residual filtering*, since it operates on the remaining (un-indexed) portion of the vector.

B. Index Construction

Lines 3 and 7-14 in Algorithm 1 highlight the index size reduction via *index filtering* in `AllPairs`. The algorithm does not start indexing feature values from $\boldsymbol{x}$ until the variable b_1 reaches the similarity threshold t. Once a value is indexed, it is erased form $\boldsymbol{x}$ (line 12). Bayardo et. al [3] show that enough features will be indexed using this method to ensure that any vector $\boldsymbol{y}$ that has the potential to meet the similarity threshold t against $\boldsymbol{x}$ will be identified during the similarity search. While a certain column order is not necessary, sorting D's columns in decreasing frequency order (line 3) ensures only less frequent features are indexed, leading to a smaller index size.

Algorithm 1 The AllPairs Algorithm

```
 1: function AllPairs($D, t$)
 2:     Reorder $D$ rows in decreasing rw order
 3:     Reorder $D$ columns in decreasing frequency order
 4:     $O \leftarrow \varnothing,\ I_j \leftarrow \varnothing,$ for $j = 1, \ldots, m$
 5:     for each $x = 1, \ldots, n$ do
 6:         $O \leftarrow O \cup \text{FindMatchesAP}(\boldsymbol{x}, \mathcal{I}, t)$
 7:         $b_1 \leftarrow 0$
 8:         for each $j = 1, \ldots, m$, s.t. $x_j > 0$ do
 9:             $b_1 \leftarrow b_1 + x_j \times \min(cw_j, rw_x)$
10:             if $b_1 \geq t$ then
11:                 $I_j \leftarrow I_j \cup \{(x, x_j)\}$
12:                 $x_j \leftarrow 0$
13:             end if
14:         end for
15:     end for
16: return O
```

The variable b_1, which we call the `pscore` (prefix score), captures an upper bound on the similarity score attainable by matching the first features in $\boldsymbol{x}$ against any other vector in the dataset. It is akin to the similarity of $\boldsymbol{x}$ with the maximum possible valued vector in the dataset, which should be computed as $b_1 \leftarrow \sum_{j=1}^{m} cw_j \times x_j$. `AllPairs` takes advantage of an imposed order on the rows of D to improve this bound. By ordering the dataset in decreasing order of *rw* (line 2), one obtains a sharper estimate on a candidate's feature value. The vectors we are interested in, which are those that follow $\boldsymbol{x}$ in the dataset, are thus guaranteed to have the maximum value for feature j of $\min(cw_j, rw_x)$.

Awekar and Samatova focus on candidate pruning in `APT`, and make no changes to the index reduction proposed in `AllPairs`. Lee et al., however, achieve better index reduction in `MMJoin` by using the non-negativity of the square of a real number property, $(a-b)^2 \geq 0 \Rightarrow a^2 + b^2 \geq 2ab$. Using this inequality, they derive

$$\text{dot}(\boldsymbol{x},\boldsymbol{y}) = \sum_j x_j \times y_j \le \sum_j \frac{{x_j}^2 + {y_j}^2}{2} = \frac{1}{2}||\boldsymbol{x}||^2 + \frac{1}{2}||\boldsymbol{y}||^2, \quad (1)$$

that also holds for prefixes or suffixes of vectors at a common feature p, i.e.:

$$\text{dot}(\boldsymbol{x}'_p, \boldsymbol{y}'_p) \le \frac{1}{2}||\boldsymbol{x}'_p||^2 + \frac{1}{2}||\boldsymbol{y}'_p||^2. \quad (2)$$

As all vectors are unit length normalized, the dot-product of $\boldsymbol{x}$ with any other vector can then be approximated by $\text{dot}(\boldsymbol{x},\cdot) \le \frac{1}{2}||\boldsymbol{x}||^2 + \frac{1}{2}$, which provides another upper bound for the pscore. MMJoin combines this new bound with the original one in AllPairs by using the minimum of the two upper bounds, $\min(b_1, b_2) \ge t$ in line 10 of Algorithm 1, where $b_2 = \frac{1}{2}||\boldsymbol{x}'_{j+1}||^2 + \frac{1}{2}$. Lee et al. also use Equation 2 during candidate generation and verification, and thus store the value $\frac{1}{2}||\boldsymbol{x}'_j||^2$, in addition to x_j, for each indexed term (line 11 becomes $I_j \leftarrow I_j \cup \{(x, x_j, \frac{1}{2}||\boldsymbol{x}'_j||^2)\}$).

Note that our explanation and notation of Lee et al.'s algorithm has been adjusted to follow the column ordering in AllPairs. Their original presentation follows the opposite column ordering. Therefore, they initially pre-compute $b_1 \leftarrow \sum_{j=1}^m x_j \times \min(cw_j, rw_x)$ in line 7, and then roll back the computation, indexing until b_1 falls below t. We have found that this step slows down the overall algorithm performance if applied to AllPairs.

Algorithm 2 AllPairs FindMatches

```
1: function FindMatchesAP($\boldsymbol{x}, \mathcal{I}, t$)
2:     $A \leftarrow \varnothing$                                  ▷ accumulator array
3:     $M \leftarrow \varnothing$                                  ▷ set of matches
4:     $sz_1 \leftarrow t/rw_x$
5:     $rs_1 \leftarrow \sum_{j=1}^m x_j \times cw_j$
6:     for each $j = m, \dots, 1$, s.t. $x_j > 0$ do
7:         $I_j \leftarrow I_j \setminus \{(y, y_j)\}, \ \forall\, y$ s.t. $|\boldsymbol{y}| < sz_1$
8:         for each $(y, y_j) \in I_j$ do
9:             if $A[y] > 0$ or $rs_1 \ge t$ then
10:                $A[y] \leftarrow A[y] + x_j \times y_j$
11:            end if
12:        end for
13:        $rs_1 \leftarrow rs_1 - x_j \times cw_j$
14:    end for
15:    for each $y$ s.t. $A[y] > 0$ do
16:        if $A[y] + \min(|\boldsymbol{x}|, |\boldsymbol{y}'|) \times rw_x \times rw'_y \ge t$ then
17:            $s \leftarrow A[y] + \text{dot}(\boldsymbol{x}, \boldsymbol{y}')$
18:            if $s \ge t$ then
19:                $M \leftarrow M \cup \{(x, y, s)\}$
20:            end if
21:        end if
22:    end for
23: return M
```

C. Candidate Generation

Candidate generation and verification in AllPairs are detailed in Algorithm 2. AllPairs uses a lower bound (sz_1), which we call minsize, to eliminate unpromising indexed vectors that are too short (lines 4 and 7). Bayardo et al. name this process *size filtering*. They show that any candidate vector must have at least t/rw_x non-zero values to possibly achieve t similarity with $\boldsymbol{x}$. Additionally, since the dataset rows are ordered in decreasing *rw* order, the minimum candidate size increases monotonically with each iteration. Those vectors that fail this check will then fail it for all future row vectors and can be safely removed from the inverted index (line 7).

APT and MMJoin both provide stronger bounds for the minsize. Awekar and Samatova use an upper bound on the dot-product, $\text{dot}(\boldsymbol{x},\boldsymbol{y}) \le rw_x \times \sum_{j=1}^m y_j$, to derive minsize as $sz_2 \le (t/rw_x)^2$. On the other hand, Lee et al. use the upper bound

$$\text{dot}(\boldsymbol{x},\boldsymbol{y}) \le \min(|\boldsymbol{x}|, |\boldsymbol{y}|) \times rw_x \times rw_y$$

to drive it as $sz_3 \le t/(rw_x \times rw_y)$.

Residual filtering uses an upper bound on the similarity of the un-indexed portion of the vectors, along with the already accumulated dot-product, to prune additional potential candidates. As we accumulate over the features of $\boldsymbol{x}$, there comes a point when there are not enough features left to allow any new vector without accumulated weight to reach the similarity threshold. AllPairs finds this point by maintaining an upper bound remscore value (rs_1) on the similarity score that a non-accumulated vector $\boldsymbol{y}$ could achieve with $\boldsymbol{x}$ (lines 5 and 13). Accumulation only starts as long as the remscore is still above the threshold t (line 9). Once accumulation has started for a vector, it becomes a *candidate*.

APT uses the same remscore bound as in AllPairs. MMJoin capitalizes on Equation 2 in two ways to enhance residual filtering. First, it augments the remscore bound in line 9 by checking $\min(rs_1, rs_2) \ge t$, where $rs_2 = \frac{1}{2}||\boldsymbol{x}'_{j+1}||^2 + \frac{1}{2}$. Note that accumulation occurs from left to right. If the vector $\boldsymbol{y}$ has not started accumulating, and $rs_2 < t$, the similarity value cannot possibly pass the threshold t, and the potential candidate is skipped. Second, for those candidates that have started accumulating, MMJoin pushes a verification step into the candidate generation stage. It keeps checking, after each accumulation change, whether $A[y] + \frac{1}{2}||\boldsymbol{x}'_j||^2 + \frac{1}{2}||\boldsymbol{y}'_j||^2$ is below the threshold t. When this estimate falls below t, MMJoin stops accumulating $\boldsymbol{y}$ and sets $A[y] = 0$. Lee et. al. call this process *length filtering*.

D. Candidate Verification

The similarity $\text{sim}(\boldsymbol{x},\boldsymbol{y})$ has already been partially computed and stored in the accumulator $A[y]$. AllPairs then tries to estimate the similarity of $\boldsymbol{x}$ with the un-indexed prefix of each candidate $\boldsymbol{y}'$ (line 16). This bound, which we call the dpscore (dot-product score), allows skipping the full similarity score computation of $\boldsymbol{x}$ with the candidate if the estimate is still below t. Otherwise, AllPairs computes the remaining similarity between $\boldsymbol{x}$ and the prefix $\boldsymbol{y}'$ exactly and adds the pair to the result M as necessary (lines 18-20).

Lee et al. employ the same dpscore bound as Bayardo et al. Leveraging the dot-product upper bound they considered in the minsize estimation, Awekar and Samatova propose a new dpscore, which they prove is a tighter bound than that of Bayardo et al., and is given by:

$$\text{dot}(\boldsymbol{x}, \boldsymbol{y}) \le A[y] + \min(rw_x \times \Sigma'_y, rw'_y \times \Sigma_x).$$

As an alternate means of pruning, based on their *length filtering* idea, Lee et al. use the prefix similarity estimates they stored in the index and check whether $A[y]+\frac{1}{2}||\boldsymbol{x}'_j||^2+\frac{1}{2}||\boldsymbol{y}'_j||^2$ drops below the threshold, even while computing the rest of the dot-product. To alleviate excessive checking, they only test this bound at every non-consecutive feature common to $\boldsymbol{x}$ and $\boldsymbol{y}$. The candidate $\boldsymbol{y}$ is pruned if the bound falls below t.

V. L2AP

We now present our algorithm, L2AP, which leverages the Cauchy-Schwarz inequality to obtain tighter ℓ^2-norm similarity estimate bounds for both index reduction and candidate generation and verification. In addition, L2AP improves on and introduces new residual filtering techniques that help eliminate the majority of candidates before fully computing their similarity value.

A. ℓ^2-norm Bounds

The majority of the improvement in L2AP is due to much tighter bounds obtained by leveraging the Cauchy-Schwarz inequality in partial dot-product estimations. Recall that, $\text{dot}(\boldsymbol{x}, \boldsymbol{y}) = \text{dot}(\boldsymbol{x}', \boldsymbol{y}) + \text{dot}(\boldsymbol{x}'', \boldsymbol{y})$, where $\boldsymbol{x}'$ is the prefix, or un-indexed portion of the vector, and $\boldsymbol{x}''$ is its suffix. By the Cauchy-Schwarz inequality we have that:

$$\text{dot}(\boldsymbol{x}', \boldsymbol{y}) \le ||\boldsymbol{x}'|| \times ||\boldsymbol{y}||. \tag{3}$$

Since all vectors are unit length normalized, the prefix dot-product can then be approximated by $\text{dot}(\boldsymbol{x}', \boldsymbol{y}) \le ||\boldsymbol{x}'||$. This new bound has profound consequences during indexing and candidate generation. Vectors are accumulated from right to left. If $||\boldsymbol{x}'|| < t$, no terms in $\boldsymbol{x}'$ can lead to new candidates that have not yet been identified.

The ℓ^2-norm bound is tighter than the one proposed by Lee et al., $\text{dot}(\boldsymbol{x}', \boldsymbol{y}) \le \frac{1}{2}||\boldsymbol{x}'||^2 + \frac{1}{2}$, since

$$(||\boldsymbol{x}'|| - 1)^2 \ge 0 \;\Rightarrow\; \frac{1}{2}||\boldsymbol{x}'||^2 + \frac{1}{2} \ge ||\boldsymbol{x}'||.$$

Their estimate will always exceed 0.5, while ours is closer to the true dot-product.

Similarly, an estimate for the dot-product of the prefixes of $\boldsymbol{x}$ and $\boldsymbol{y}$ at a common term j is given by,

$$\text{dot}(\boldsymbol{x}'_j, \boldsymbol{y}'_j) \le ||\boldsymbol{x}'_j|| \times ||\boldsymbol{y}'_j||. \tag{4}$$

Candidates can be pruned at a common term j if the sum of their accumulated score and this prefix dot-product estimate falls below the threshold t. Again, this bound is tighter than the similar bound proposed by Lee et al., $\text{dot}(\boldsymbol{x}', \boldsymbol{y}') \le \frac{1}{2}||\boldsymbol{x}'||^2 + \frac{1}{2}||\boldsymbol{y}'||^2$, since

$$(||\boldsymbol{x}'|| - ||\boldsymbol{y}'||)^2 \ge 0 \;\Rightarrow\; \frac{1}{2}||\boldsymbol{x}'||^2 + \frac{1}{2}||\boldsymbol{y}'||^2 \ge ||\boldsymbol{x}'|| \times ||\boldsymbol{y}'||.$$

B. Index Construction

Algorithm 3 delineates our proposed method, L2AP. We will now highlight the improvements we introduce over the AP framework we discussed in Section IV.

Algorithm 3 The L2AP Algorithm

```
1: function L2AP($D, t$)
2:     Reorder $D$ rows in decreasing $rw$ order
3:     Reorder $D$ columns in decreasing frequency order
4:     $O \leftarrow \varnothing$; $I_j \leftarrow \varnothing$; $\hat{cw}_j \leftarrow 0$, for $j = 1, \ldots, m$
5:     for each $x = 1, \ldots, n$ do
6:         $O \leftarrow O \cup$ FindMatchesL2AP($\boldsymbol{x}, \mathcal{I}, ps, \hat{cw}, t$)
7:         $b_1 \leftarrow 0$; $b_t \leftarrow 0$; $b_3 \leftarrow 0$
8:         for each $j = 1, \ldots, m$, s.t. $x_j > 0$ do
9:             $pscore \leftarrow \min(b_1, b_3)$
10:            $b_1 \leftarrow b_1 + x_j \times \min(cw_j, rw_x)$
11:            $b_t \leftarrow b_t + x_j^2$; $b_3 \leftarrow \sqrt{b_t}$
12:            if $\min(b_1, b_3) \ge t$ then
13:                $ps[x] \leftarrow pscore$ if $ps[x] = 0$
14:                $I_j \leftarrow I_j \cup \{(x, x_j, ||\boldsymbol{x}'_j||)\}$
15:                $x_j \leftarrow 0$
16:            end if
17:        end for
18:    end for
19: return O
```

We improve the pscore bound in the AllPairs framework using our tighter ℓ^2-norm bound. The variable b_3 computes $||\boldsymbol{x}'_{j+1}||$, the ℓ^2-norm of the prefix of $\boldsymbol{x}$ ending at index j, inclusive. As shown in Section V-A, no new candidates can be identified during accumulation once the prefix norm $||\boldsymbol{x}'_j||$ falls below t. To postpone indexing further, we use the lesser of our new bound, b_3, and the bound proposed by Bayardo et al., to find the minimum number of features we must index. Additionally, we store the prefix ℓ^2-norm $||\boldsymbol{x}'_j||$ in the index (line 14), to be used during the candidate generation and verification stages of the algorithm.

The pscore bound estimates the similarity of $\boldsymbol{x}'$ with any other vector in the dataset. We store the pscore value for $\boldsymbol{x}'_j$ (lines 9 and 13) and use it during candidate verification as an effective pruning strategy for false positive candidates.

C. Candidate Generation

Candidate generation and verification in L2AP are detailed in Algorithm 4. L2AP uses the same minsize upper bound as in MMJoin, $sz_3 \le t/(rw_x \times rw_y)$, which is a better bound than the respective one in APT, $sz_2 \le t^2/rw_x^2$. Note that $t^2 \le t$, since $t \in [0, 1]$. Given the decreasing rw order of the dataset, $rw_y \ge rw_x$, and $rw_x^2 \le rw_x \times rw_y$. It follows that $sz_3 \ge sz_2$.

The remscore bound enables our algorithm to stop adding new candidates once the estimated dot-product between the prefix of $\boldsymbol{x}$ and all possible candidates falls below t. We improve this bound in two ways. First, note that the similarity of $\boldsymbol{x}$ is computed only against vectors in the inverted index, which come before it in dataset processing order. We use a tighter feature maximum value, $\hat{cw}_j$, in rs_3, an *enhanced* version of Bayardo's proposed remscore bound (line 4), which is computed only over those vectors in the inverted

Algorithm 4 L2AP FindMatches

```
1:  function FindMatchesL2AP($\boldsymbol{x}, \mathcal{I}, ps, \hat{cw}, t$)
2:      $A \leftarrow \varnothing;\ M \leftarrow \varnothing$
3:      $sz \leftarrow t / rw_x$
4:      $rs_3 \leftarrow \sum_{j=1}^{m} x_j \times \hat{cw}_j;\ rs_t \leftarrow 1;\ rs_4 \leftarrow 1$
5:      $rw'_{x_j} \leftarrow \max(\boldsymbol{x}'_j),\ \Sigma'_{x_j} \leftarrow \sum_{i=1}^{j-1} x_i,\ \forall\, x_j > 0$
6:      for each $j = m, \ldots, 1$, s.t. $x_j > 0$ do
7:          $I_j \leftarrow I_j \setminus \{(y, y_j, ||\boldsymbol{y}'_j||)\},\ \forall\, y$ s.t. $|\boldsymbol{y}| \times rw_y < sz$
8:          for each $(y, y_j, ||\boldsymbol{y}'_j||) \in I_j$ do
9:              if $A[y] > 0$ or $\min(rs_3, rs_4) \ge t$ then
10:                 $A[y] \leftarrow A[y] + x_j \times y_j$
11:                 if $A[y] + ||\boldsymbol{x}'_j|| \times ||\boldsymbol{y}'_j|| < t$ then
12:                     $A[y] \leftarrow 0$
13:                 end if
14:             end if
15:         end for
16:         $rs_3 \leftarrow rs_3 - x_j \times \hat{cw}_j$
17:         $rs_t \leftarrow rs_t - x_j^2;\ rs_4 \leftarrow \sqrt{rs_t}$
18:     end for
19:     for each $y$ s.t. $A[y] > 0$ do
20:         next $y$ if $A[y] + ps[y] < t$
21:         next $y$ if $A[y] + \min(rw_x \times \Sigma'_y, rw'_y \times \Sigma_x) < t$
22:         find $p$ s.t. $y_p \in \boldsymbol{y}' \wedge y_p > 0 \wedge x_p > 0$
23:         $s \leftarrow A[y] + x_p \times y_p$
24:         next $y$ if $s + \min(rw'_{x_p} \times \Sigma'_{y_p}, rw'_y \times \Sigma'_{x_p}) < t$
25:         for each $j > p$ s.t. $y_j > 0 \wedge x_j > 0$ do
26:             $s \leftarrow s + x_j \times y_j$
27:             if $s + ||\boldsymbol{x}'_j|| \times ||\boldsymbol{y}'_j|| < t$ then
28:                 next $y$
29:             end if
30:         end for
31:         if $s \ge t$ then
32:             $M \leftarrow M \cup \{(x, y, s)\}$
33:         end if
34:     end for
35:     $\hat{cw}_j \leftarrow \max(x_j, \hat{cw}_j),\ \forall\, x_j > 0$
36: return M
```

index. Each $\hat{cw}_j$ is updated to the new maximum value after completing the current search (line 35).

Our second improvement involves the ℓ^2-norm bound we discussed in Section V-A. The variable rs_4 uses Equation 3 to estimate the dot-product of $\boldsymbol{x}'_{j+1}$, the prefix of $\boldsymbol{x}$ ending at term j, inclusive, with any other vector. As long as $||\boldsymbol{x}'_{j+1}||$ is not below our threshold t, we can start accumulating a similarity value for a new candidate (line 9).

We use the lesser of rs_3 and rs_4 for our `remscore` bound. While rs_3 can at times be a tighter bound than rs_4, we estimate that most of the time rs_4 will provide a better prefix similarity estimate. At some index p, the two bounds are computed as $rs_3^p = \sum_{j=1}^{p} x_j \times \hat{cw}_j$ and $rs_4^p = \sqrt{\sum_{j=1}^{p} x_j \times x_j}$. For most values, $\hat{cw}_j \gg x_j$, especially given the decreasing *rw* ordering of the vectors, which will likely lead to $rs_3^p > rs_4^p$.

Similar to `MMJoin`, we push a verification step into the candidate generation portion of our algorithm. Based on Equation 4, after each accumulation operation, we check whether the estimated prefix similarity, $\text{dot}(\boldsymbol{x}', \boldsymbol{y}') = ||\boldsymbol{x}'_j|| \times ||\boldsymbol{y}'_j||$, will be enough to push the score already accumulated over the threshold (line 11). If this check fails, we cease accumulating $\boldsymbol{y}$ and move to the next candidate.

D. *Candidate Verification*

We introduce a new type of candidate pruning, based on the `pscore` bound we computed during indexing, which we call *`pscore` filtering*. At the end of the candidate generation stage, the accumulator $A[y]$ contains a partial dot-product, $\text{dot}(\boldsymbol{x}, \boldsymbol{y}'')$. Recall that the `pscore` bound estimated the dot-product between the prefix of $\boldsymbol{y}$ and any other vector in the dataset, $\text{dot}(\boldsymbol{y}', \cdot)$. We stored this estimate at the end of indexing $\boldsymbol{y}$ and use it here for candidate verification, for an estimate of $\text{dot}(\boldsymbol{y}', \boldsymbol{x})$. If the sum of the accumulated score and the estimate falls below t, the candidate is discarded (line 20).

We adopt the `dpscore` bound introduced by Awekar and Samatova, and provide several enhancements, similar in spirit to the *positioning filtering* idea of Xiao et al [1]. We efficiently compute the dot-product of $\boldsymbol{x}$ with candidates by pre-hashing $\boldsymbol{x}$'s values. In addition, we choose to also hash prefix maximum and prefix sum values of $\boldsymbol{x}$ at each position j where $x_j > 0$ (line 5), which aid in strengthening the `dpscore` bound. Once the first common feature p is found between $\boldsymbol{x}$ and $\boldsymbol{y}'$ (line 22), the following possible `dpscore` variants can be used,

$$\text{dot}(\boldsymbol{x}, \boldsymbol{y}) \le A[y] + \min(rw_x \times \Sigma'_y, rw'_y \times \Sigma_x), \quad (5)$$
$$\text{dot}(\boldsymbol{x}, \boldsymbol{y}) \le A[y] + \min(rw'_{x_p} \times \Sigma'_{y_p}, rw'_y \times \Sigma_x), \quad (6)$$
$$\text{dot}(\boldsymbol{x}, \boldsymbol{y}) \le A[y] + \min(rw_x \times \Sigma'_{y_p}, rw'_y \times \Sigma'_{x_p}), \quad (7)$$
$$\text{dot}(\boldsymbol{x}, \boldsymbol{y}) \le A[y] + \min(rw'_{x_p} \times \Sigma'_{y_p}, rw'_y \times \Sigma'_{x_p}). \quad (8)$$

Similar enhancements are possible for Bayardo's `dpscore` bound. By hashing prefix maximum and prefix size values in addition to the values of $\boldsymbol{x}$, we can utilize the following bounds once the first common feature p between $\boldsymbol{x}$ and $\boldsymbol{y}'$ is found,

$$\text{dot}(\boldsymbol{x}, \boldsymbol{y}) \le A[y] + \min(|\boldsymbol{x}|, |\boldsymbol{y}'|) \times rw_x \times rw'_y, \quad (9)$$
$$\text{dot}(\boldsymbol{x}, \boldsymbol{y}) \le A[y] + \min(|\boldsymbol{x}|, |\boldsymbol{y}'_p|) \times rw'_{x_p} \times rw'_{y_p}, \quad (10)$$
$$\text{dot}(\boldsymbol{x}, \boldsymbol{y}) \le A[y] + \min(|\boldsymbol{x}'_p|, |\boldsymbol{y}'_p|) \times rw_x \times rw'_{y_p}, \quad (11)$$
$$\text{dot}(\boldsymbol{x}, \boldsymbol{y}) \le A[y] + \min(|\boldsymbol{x}'_p|, |\boldsymbol{y}'_p|) \times rw'_{x_p} \times rw'_{y_p}. \quad (12)$$

Note that Equations 5 and 9 can be used before finding the first common feature p. One could also try the cheaper bound in Equation 5 or Equation 9 (line 21), followed by one of the position-based bounds in case of failure (line 24). Equations 8 and 12 provide the tightest bounds among their respective variants, since $rw'_{x_p} \le rw_x$, $\Sigma'_{x_p} \le \Sigma_x$, and $|\boldsymbol{x}'_p| \le |\boldsymbol{x}|$. A similar proof as provided by Awekar and Samatova (Section 4.4 in [18]), showing that Equation 5 provides a better bound than Equation 9, can be constructed to show the superiority of Equation 8 for candidate pruning, making it the best of the eight proposed `dpscore` bounds.

If a candidate passes these initial checks, we compute the full dot-product of its remaining prefix with the query vector (lines 25-30). After each accumulation, however, we use our ℓ^2-norm based prefix similarity estimate to further prune unpromising candidates (lines 27-29). Surviving candidates have their final similarity value checked against the threshold t and are added to the result M if they still exceed it.

E. L2AP-approx

Using Bayesian inference, BayesLSH finds the probability that the similarity of two candidates is above the threshold t, conditional on the observed event of LSH hash matches. Additionally, it can estimate the similarity value and the probability that the estimate is within δ of the true similarity. Satuluri and Parthasarathy provide a way to tractably perform inference for the Jaccard and cosine similarity functions applied to both binary and weighted vectors. BayesLSH-Lite is a less expensive variant of the algorithm that, after examining a fixed number of hashes h, uses the first probability estimate to prune candidates, with a theoretically guaranteed maximum false negative rate of ϵ. It then computes the exact similarity value for unpruned candidate pairs. This strategy has been proven effective, outperforming both AllPairs and LSH. The remaining details of BayesLSH and BayesLSH-Lite are beyond the scope of this paper and can be found in [21].

We are interested in exact similarity values in our problem, so we combine our algorithm with BayesLSH-Lite to form an approximate variant, L2AP-approx. In Algorithm 4, we replace ℓ^2-norm based candidate verification (lines 25-30) with BayesLSH-Lite filtering applied to a pair of vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ (lines 6-14 in Algorithm 2 of [21]). We then complete accumulation and similarity threshold checking for unpruned candidates. Using BayesLSH-Lite at this point allows us to take advantage of most of our pruning strategies before resorting to approximate estimation. While L2AP-approx may over-prune in this step, it will provide exact similarity values for the neighbors it finds.

F. Discussion

Our strategy, so far, has been to improve and provide new similarity bounds that can lead to better index reduction, candidate generation, and candidate pruning. Many of the bounds we proposed come with the added cost of more hashing or bound computations. In some cases, this cost may outweigh the benefit of a somewhat smaller candidate set or fewer full dot-products being computed. For example, using Equation 6 or 7 instead of 8 for the dpscore bound has the benefit of less hashing, while still being a tighter bound than the one in Equation 5. With these thoughts in mind, we built our prototypes, L2AP and L2AP-approx, with the ability to choose, at compile time, the pruning strategies to employ. This gives us the added benefit of being able to check the effectiveness of individual pruning bounds. Table I summarizes some of the choices available for each bound in our prototype. The symbols b_x, sz_x, and rs_x refer to the respective pscore, minsize, and remscore bounds described in this paper. We use $dp_1 - dp_8$ to reference the dpscore pruning choices in Equations 5 through 12. The index construction stage is noted as *i.c.* We note ℓ^2-norm filtering at the candidate generation (*c.g.*) and verification (*c.v.*) stages of the algorithm as $l2cg$ and $l2cv$, respectively. We will use this notation to specify pruning strategies in Section VI.

TABLE I
PRUNING STRATEGIES IN L2AP

Stage	Bound	Bound Choices
i.c.	pscore	b_1, $\min(b_1, b_2)$, $\min(b_1, b_3)$
c.g.	minsize	sz_1, sz_3
	remscore	rs_1, rs_2, rs_3, rs_4, $\min(rs_1, rs_2), \min(rs_1, rs_4), \min(rs_3, rs_4)$
	ℓ^2-norm	$l2cg$
c.v.	pscore	ps
	dpscore	$\{dp_1, dp_5\}$+$\{dp_2, dp_3, dp_4, dp_6, dp_7, dp_8\}$, $dp_1 \to dp_8$
	ℓ^2-norm	$l2cv$

VI. Experimental Evaluation

We leveraged the modular nature of our prototype, L2AP, to test the effectiveness of our proposed new similarity estimate bounds on 6 real datasets. Additionally, we evaluated the performance of L2AP and L2AP-approx against state-of-the-art exact and approximate baseline approaches, AllPairs, MMJoin, and BayesLSH-Lite, measuring the full execution time for APSS at a wide range of similarity thresholds.

TABLE II
DATASET STATISTICS

Dataset	n	m	nnz	mrl	mcl
RCV1	804414	43001	61e6	76	1417
WikiWords500k	494244	343622	197e6	399	574
WikiWords100k	100528	339944	79e6	787	233
TwitterLinks	146170	143469	200e6	1370	1395
WikiLinks	1815914	1648879	44e6	24	27
OrkutLinks	3072626	3072441	223e6	73	73

A. Datasets

We use the same 6 datasets to evaluate our prototypes as were used in [21]. They represent some real-world networks and benchmark text corpora popularly used in text-categorization research. Their characteristics, including number of rows (n), columns (m), and non-zeros (nnz), and mean row/column length (mrl/mcl), are detailed in Table II. Both link and text-based datasets are represented as *tf-idf* weighted vectors. We present additional details below.

- **RCV1** is a standard benchmark corpus containing over 800,000 newswire stories provided by Reuters, Ltd. for research purposes, made available by Lewis et al. [23].
- **WikiWords500k** was kindly provided to the authors by Satuluri and Parthasarathy [21], along with the WikiWords100k and WikiLinks datasets. It contains documents with at least 200 distinct features, extracted from the September 2010 article dump of the English Wikipedia[1] (Wiki dump).
- **WikiWords100k** contains documents from the Wiki dump with at least 500 distinct features.
- **TwitterLinks**, first provided by Kwak et al. [24], contains *follow* relationships of a subset of Twitter users that follow at least 1,000 other users.

[1]http://download.wikimedia.org

- **WikiLinks** represents a directed graph of hyperlinks between Wikipedia articles in the Wiki dump.
- **OrkutLinks** contains the friendship network of over 3M users of the Orkut social media site, made available by Mislove et al. [25].

B. Baseline Approaches

We compare L2AP and L2AP-approx against the following baseline approaches.

1) IdxJoin is a straight-forward baseline that first builds a full inverted index. Then, without performing any pruning, it uses the index to compute exactly the similarity of each vector with all preceding vectors in the dataset.
2) AllPairs is a state-of-the-art approach for solving the APSS problem proposed by Bayardo et al [3], which we detailed in Section IV.
3) MMJoin enhances AllPairs by adding *length filtering* and a tighter minsize bound. These enhancements are also detailed in Section IV.
4) AllPairs+BayesLSH-Lite and LSH+BayesLSH-Lite are state-of-the-art approximate methods proposed by Satuluri and Parthasarathy [21], variants of BayesLSH that take as input the candidate set generated by AllPairs and LSH, respectively. They have been shown to significantly outperform LSH, which we do not include in the comparison.

The BayesLSH package[2] includes implementations for LSH, AllPairs+BayesLSH-Lite, LSH+BayesLSH-Lite, and AllPairs. An implementation of MMJoin was not available. We implemented IdxJoin, AllPairs[3], MMJoin, L2AP, and L2AP-approx[4] in C. Our method and all baselines are single-threaded, serial programs. Each method was executed on its own node in a cluster of HP ProLiant BL280c G6 blade servers, each with 2.8 GHz Intel Xeon processors and 24 Gb RAM. Methods that took longer than 48 hours to execute were terminated. For each method, we varied the similarity threshold between 0.3 and 0.95, in increments of 0.05. To further qualify the utility of our method for near-duplicate object detection, we also executed each method for similarities between 0.96 and 0.99, in increments of 0.01. As suggested by Satuluri and Parthasarathy, we used $\epsilon = 0.03$ (97% recall) and checked $h = 128$ hashes in both BayesLSH-Lite and L2AP-approx approximate pruning. For approximate methods, we executed each test a minimum of three times and report the average time over all test executions.

C. Effectiveness Testing

In this section, we show the effectiveness of ℓ^2-norm filtering and provide some guidance for other pruning choices.

[2]See http://www.cse.ohio-state.edu/~satuluri/research.html

[3]Unlike the AllPairs implementations by Bayardo et al. and in the BayesLSH package, ours uses a dense representation of the query vector. We found our implementation, on average, to be 2.5x faster than the one in the BayesLSH package and use it as baseline in this paper. This implementation detail has since been incorporated into the other packages.

[4]Source code for all methods is available at http://cs.umn.edu/~dragos/l2ap

Due to space limitations, we do not show results for some datasets in this section if they follow the same trend as those presented. All figures are best viewed in color.

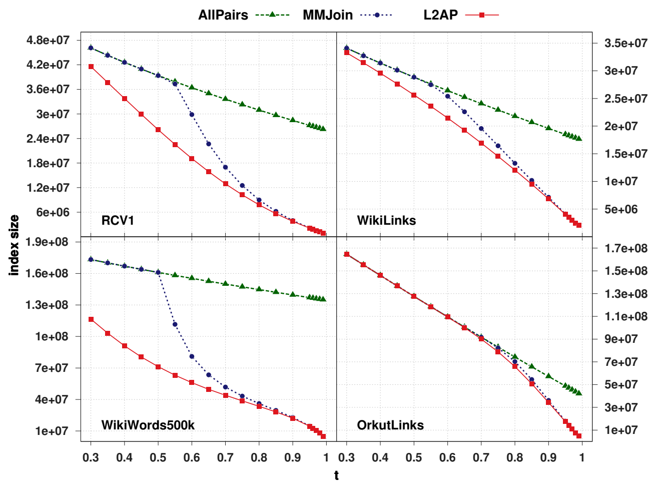

Fig. 1. Index size reduction in L2AP vs. previous methods.

1) Effectiveness of the New pscore Bound for Index Reduction: A smaller pscore bound at each threshold t leads to Algorithms 1 and 3 starting the indexing process later, thus smaller inverted indexes. We proposed that our ℓ^2-norm based prefix similarity estimate is more effective at lowering this bound than previous strategies. Figure 1 shows the index sizes achieved using the pscore bound in L2AP, MMJoin, and the original bound in AllPairs. As expected, the pscore bound in L2AP produces significantly smaller indexes than previous bounds. While the bound in MMJoin achieves similar index sizes at high values for t, it degrades to the performance of the AllPairs bound as $t \to 0.5$. OrkutLinks is the only dataset for which our ℓ^2-norm bound is unable to reduce the index size further than the pscore bound in AllPairs, for $t < 0.7$.

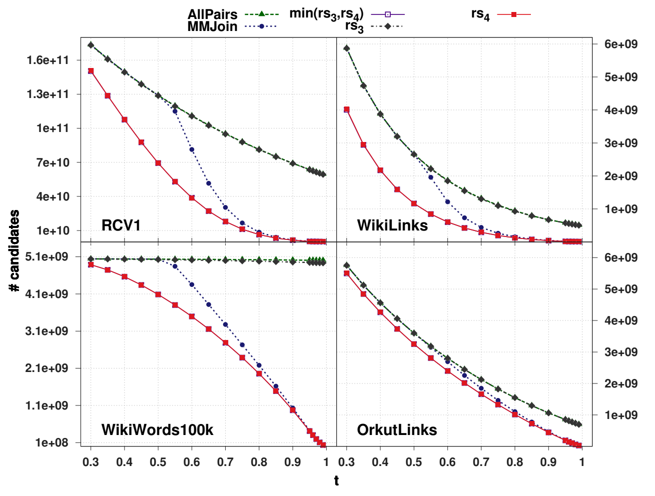

Fig. 2. Candidate pool sizes when using diverse remscore bounds.

2) Effectiveness of the New `remscore` *Bound for Candidate Generation:* In Section V-C, we estimated that our proposed `remscore` bound, rs_4, is tighter than our *enhanced* version of the bound proposed by Bayardo et al., rs_3. To verify this intuition, we counted, during the algorithm execution, how many times rs_3 vs. rs_4 was the minimum value in $\min(rs_3, rs_4)$ (line 9 of Algorithm 4). For this test we used parameters that minimized candidate generation $(\min(b_1, b_3), sz_3)$, and only counted when a new candidate was being generated, i.e. when $\min(rs_3, rs_4) \geq t$ and $A[y] = 0$. The results, which are detailed in Table III, confirmed our estimation. Our new bound, rs_4, was the minimum, averaged over all similarity values, over 97.6% of the time the `remscore` bound was checked. This suggests that L2AP can be effective, and possibly more efficient, using only the ℓ^2-norm part of the `remscore` bound, i.e. $rs_4 \geq t$ instead of $\min(rs_3, rs_4) \geq t$ in line 9 of Algorithm 4.

TABLE III
PERCENTAGE OF CASES WHERE rs_4 IS A LOWER BOUND THAN rs_3

Dataset	%	Dataset	%
RCV1	99.82	WikiLinks	99.53
WikiWords500k	99.47	TwitterLinks	98.75
WikiWords100k	98.05	OrkutLinks	97.61

In another test, we compared the effectiveness of the new `remscore` bounds, rs_3, rs_4 and $\min(rs_3, rs_4)$, against previous bounds rs_1 (`AllPairs`) and $\min(rs_1, rs_2)$ (`MMJoin`), by counting the number of candidates being generated when using the same indexing strategy as in `AllPairs` (b_1). We did not use any additional pruning or index reduction in this test. Figure 2 shows the candidate pool sizes achieved when using the different `remscore` bounds. The enhanced version of Bayardo's bound, rs_3 (almost covering the `AllPairs` line in the figure), is unable to reduce the number of candidates much more than the `AllPairs` bound. On the other hand, rs_4 significantly outperforms both rs_3 and the bound in `MMJoin`, resulting in significant reductions in the candidate pool size. The minimum of the two bounds, $\min(rs_3, rs_4)$, is overshadowed by rs_4 in the figure.

3) Effectiveness of the New ℓ^2-Norm Filtering: The ℓ^2-norm based similarity estimation in L2AP is the most effective of our pruning strategies. We have already shown, in Section VI-C1, that it greatly reduces the size of the inverted index being constructed. We now evaluate the effectiveness of ℓ^2-norm based pruning in the candidate generation and candidate verification stages of our algorithm. For this test, we do not use the ℓ^2-norm during index construction, leveraging only the `AllPairs` bound (b_1) for this step. We test a *baseline* with no ℓ^2-norm filtering, then add it in the candidate generation stage (*l2cg*), and in the candidate verification stage (*l2cv*) of the algorithm. We record, under each scenario, the number of unpruned dot-products and total execution time.

As can be seen in Figure 3, ℓ^2-norm filtering in L2AP is able to drastically reduce the number of dot-products being computed, at times by several orders of magnitude. We find

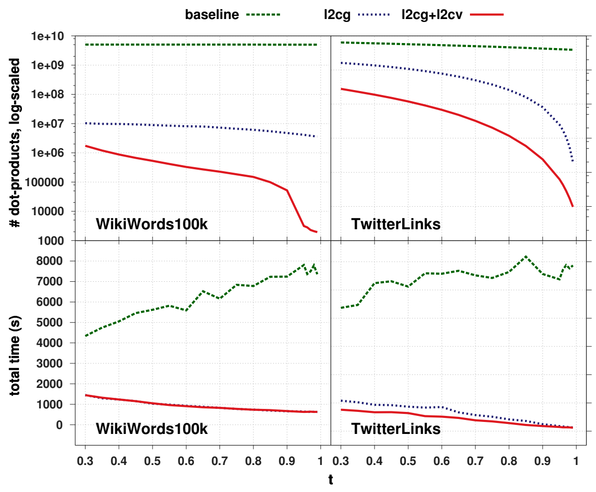

Fig. 3. Number of dot-products and total time with and w/o ℓ^2-norm filtering.

that the majority of the pruning happens in the candidate generation step, and most of the cost associated with this bound is in the initial computation of the prefix magnitude, $||x'||$, which is stored in the index or hashed. Thus, we find little difference in execution times when enabling ℓ^2-norm filtering in the c.v. stage in addition to the c.g. stage.

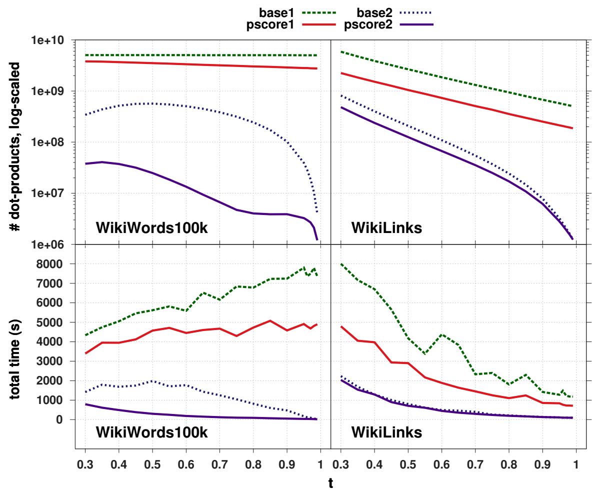

Fig. 4. Number of dot-products and total time with and w/o `pscore` filtering.

4) Effectiveness of the New `pscore` *Bound for Candidate Pruning:* We compared our algorithm's execution with and without `pscore` filtering, measuring the number of unpruned dot-products and total execution time, under two experimental scenarios. In the first, we used `AllPairs` bounds in the index reduction and candidate generation stages (b_1, rs_1), allowing `pscore` filtering to be most productive. The `pscore` in this test is based primarily on the dot-product estimate with the maximum possible vector in the dataset, and does not take advantage of the ℓ^2-norm based prefix similarity estimate. We note this baseline without `pscore` filtering as *base1* in Figure 4, and the results of this experiment with `pscore` filtering as *pscore1*. In a second experiment, we enabled the most pruning possible in the i.c. and c.g. stages of the algorithm ($\min(b_1, b_3)$, $\min(rs_3, rs_4)$, and $l2cg$), and no other pruning during the c.v. stage. The `pscore` here takes advantage of the ℓ^2-norm based prefix similarity estimate computed during indexing. We note this baseline without `pscore` filtering as *base2*, and the result with `pscore` filtering as *pscore2*.

As can be seen in Figure 4, `pscore` filtering is quite effective at reducing the number of unpruned dot-products, which results in significantly smaller execution times (up to 38% smaller). While its effectiveness is reduced when previous pruning occurs, as expected, `pscore` filtering still reduces execution time by considerable amounts for text datasets.

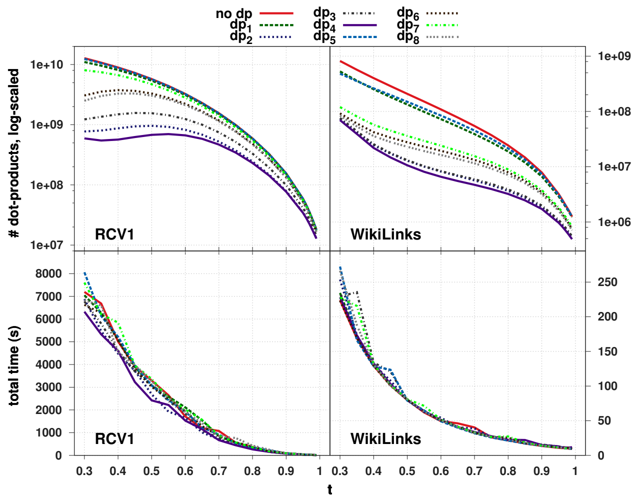

Fig. 5. Number of dot-products and total time when using diverse `dpscore` bounds.

5) Effectiveness of the New `dpscore` Bounds for Positional Filtering: Figure 5 shows the number of unpruned dot-products and total execution time when pruning using each of the `dpscore` bounds we proposed. For this test, we employed maximal index reduction and candidate generation pruning ($\min(b_1, b_3)$, $\min(rs_3, rs_4)$, and $l2cg$), and only `dpscore` pruning during candidate verification. We also included a baseline in which no `dpscore` pruning was used (*no dp*).

As predicted, dp_4 is able to achieve the best reduction in the number of unpruned dot-products. However, it requires the most hashing and can sometimes lead to longer execution times than other `dpscore` bounds. Overall, the amount of pruning achieved using the various `dpscore` bounds only leads to modest reductions in the execution time.

We also tested combinations of dp bounds, as noted in Table I. Space limitations prevent us from showing these results. Overall, we have found the best `dpscore` pruning strategy to be $dp_5 + dp_6$ for most datasets. While this strategy is not able to forego as many dot-product computations as dp_4, it does not require computing and storing vector prefix sums, a source of delay in $dp_1 - dp_4$. When testing `dpscore` pruning in concert with other filtering strategies at the candidate verification stage, we found that, for high similarity values, `dpscore` pruning is overshadowed by ℓ^2-norm and `pscore` pruning, and becomes ineffective.

6) A Word on the `minsize` Bound: Similar to Bayardo et al., we implement inverted lists as arrays and lazily remove vectors pruned by the `minsize` bound, only from the beginning of the lists. Using this strategy, we found that size filtering provided little additional pruning over the other strategies, and in most cases slowed down the overall computation. We forego presenting those results due to lack of space.

D. Efficiency Testing

We compare the total execution time of `L2AP` in relation to exact and approximate baselines. Figures are best viewed in color.

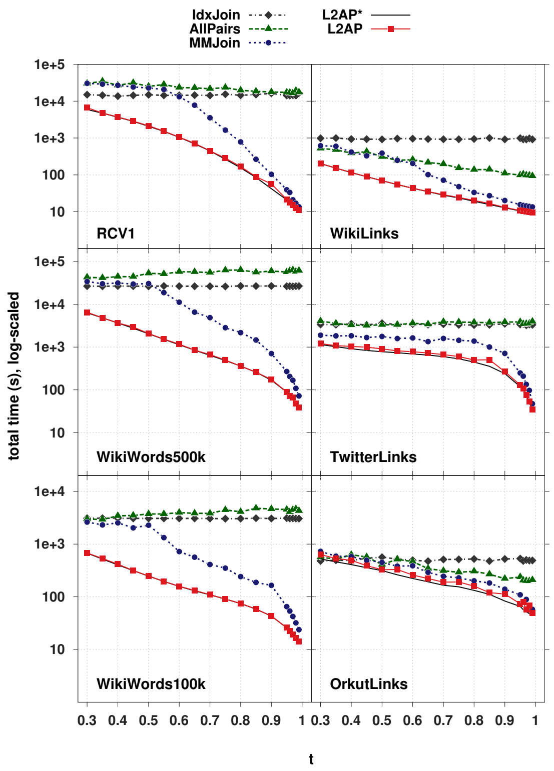

Fig. 6. Total execution times in seconds for exact algorithms.

1) Comparison with Exact Baselines: In the previous section, we noted the effectiveness of the individual pruning strategies proposed in this paper. Each pruning strategy comes with additional bounds computation and checking costs, and leads to efficient similarity search only when it is highly effective at reducing the index size, pruning candidates, or stopping accumulation early for false positive candidates. Combining strategies is not always straight-forward, as pruning in one stage of the algorithm can affect the effectiveness of bounds in later stages. We found the most efficient combination of pruning strategies across datasets and similarity thresholds to be ℓ^2-norm enhanced index construction ($\min(b_1, b_3)$), ℓ^2-norm based candidate generation (rs_4, $l2cg$), and ℓ^2-norm and dpscore filtering in the candidate verification stage (ps, dp_5, dp_6, $l2cv$). We use this pruning strategy across all datasets and similarity thresholds as representative of our algorithm, L2AP.

Figure 6 shows the total execution times for L2AP and the other exact baselines, IdxJoin, AllPairs, and MMJoin. We also include results for L2AP*, in which we choose the best performing pruning strategy for each dataset and similarity threshold combination. L2AP* will then always perform as well as or better than L2AP. However, L2AP performs almost as well as it could, given optimal pruning choices. For text datasets, the two schemes have nearly identical timings, the line for L2AP in Figure 6 almost completely hiding the one for L2AP*, and their differences are rather small for the other datasets.

L2AP is able to outperform exact baselines in most cases and achieves significant speedups, up to 1600x against AllPairs, and 2x-13x in general over the best exact baseline. Its best performance is at high similarity thresholds, showing its usefulness in tasks such as near-duplicate object detection. The most drastic performance difference is between L2AP and AllPairs or IdxJoin at $t = 0.99$. L2AP's much smaller index and effective candidate pruning strategies allow it to finish the similarity search in a few seconds, while AllPairs and IdxJoin spend hours to accomplish the same task. An interesting observation is that our straight-forward IdxJoin baseline, which does no pruning and fully computes vector similarities, outperforms AllPairs in several datasets. This shows that excessive bounds checking which does not lead to enough pruning can be detrimental in similarity search.

MMJoin uses similar index reduction and pruning strategies as L2AP, and is able to achieve comparable performance at high similarity thresholds. L2AP's *ℓ^2-norm filtering* is shown more effective than MMJoin's *length filtering*, however, especially at low similarity thresholds. While MMJoin degrades to the same efficiency as AllPairs at $t = 0.5$, L2AP is able to finish the task an order of magnitude faster for text datasets.

Link datasets present different challenges, often having much smaller vector and inverted list sizes than text datasets. This limits the effectiveness of the type of pruning that filtering APSS methods utilize. The smaller dimensionality and varied term usage within documents lead to longer inverted lists and better pruning potential in text datasets. While the speedup is not as dramatic as for text datasets, the pruning strategies in L2AP are effective for link datasets also, achieving up to 4.7x speedup. As Bayardo et al. have also noted [3], OrkutLinks has an artificial 1000 friend limit that prevents highly frequent features, leading to the least possibility of improvement for L2AP over prefix-filtering baselines.

E. Comparison with Approximate Baselines

Figure 7 gives a different view into the total time comparison, showing speedups obtained by L2AP against both exact and approximate baselines. We executed L2AP-approx with the same pruning parameters we used for L2AP, other than $l2cv$, which is replaced by BayesLSH-Lite pruning. In

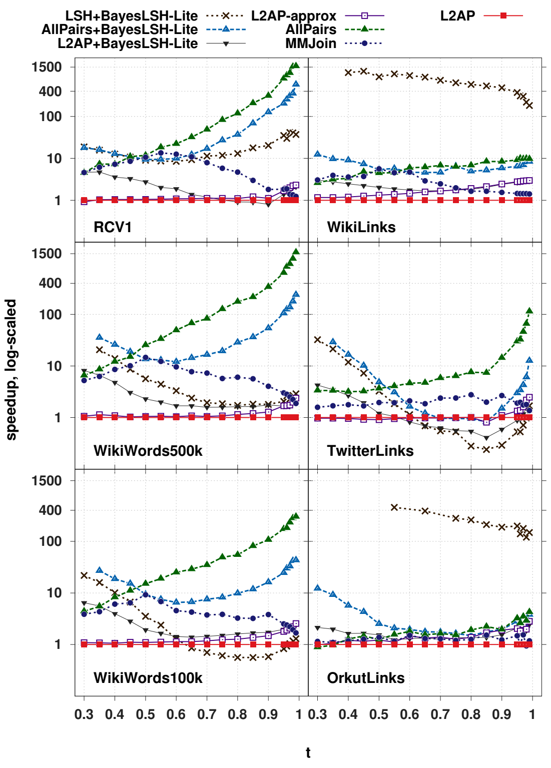

Fig. 7. L2AP speedup over competing methods.

addition, we tested a version in which L2AP was used only for candidate generation, and BayesLSH-Lite was used for candidate verification and pruning, similar to LSH+BayesLSH-Lite and AllPairs+BayesLSH-Lite. We denote this version in the Figure as L2AP+BayesLSH-Lite.

L2AP generally outperforms approximate baselines, especially at low similarity thresholds. LSH+BayesLSH-Lite outperforms L2AP only for the WikiWords100k and TwitterLinks datasets, and only at similarity values above 0.6. For other datasets, such as WikiLinks and OrkutLinks, LSH+BayesLSH-Lite was not able to finish APSS at low similarities in the time allotted (48 hours). LSH degrades quickly for high dimensional datasets and as t decreases, producing large candidate pools that cannot be pruned fast enough even by BayesLSH-Lite. In contrast, L2AP performs well for all datasets and for both high and low similarity thresholds, and returns all similar enough object pairs after the search.

It is interesting to note, looking at Figure 7, that L2AP outperforms L2AP-approx in most cases, even as their execution times are often close. L2AP is able to prune most candidates before the approximate BayesLSH-Lite candidate pruning step in L2AP-approx. The remaining pruning is not enough to outweigh the cost associated with LSH hashing or Bayesian inference in BayesLSH-Lite. L2AP+BayesLSH-Lite is able to outperform L2AP for only a few high similarity values on two of the datasets we tested. As candidate pool sizes increase, at lower similarity values, L2AP-approx is substantially slowed down by excessive hashing in BayesLSH-Lite.

VII. Conclusions and Future Work

The all-pairs similarity search problem is of utmost importance in applications such as near-duplicate object detection, clustering, and collaborative filtering. In the context of *weighted vectors* and *cosine similarity*, we fit previous methods for solving the problem in a general framework proposed by Bayardo et al. Within the framework, we introduced *ℓ^2-norm index*, *residual*, and *positional filtering*, `pscore` *filtering*, and improvements on several other existing similarity estimation bounds. These lead to drastic reductions in the inverted index size, candidate pool, and number of unpruned dot-products which must be computed fully. We proved the effectiveness of the new bounds both theoretically and experimentally, and estimated our method's efficiency against state-of-the-art baseline methods `AllPairs`, `MMJoin`, and `BayesLSH-Lite`. Our prototype, `L2AP`, was able to achieve significant speedups over all exact baselines, up to 1600x against `AllPairs`, and 2x-13x in general over the best alternative. Finally, we showed `BayesLSH-Lite` approximate candidate pruning cannot improve significantly over the exact pruning strategies introduced in `L2AP`.

Our method was proven very effective, especially at high similarity thresholds. It would be interesting to evaluate the efficiency of ℓ^2-norm filtering in the context of other similarity functions, e.g. the Dice and Tanimoto similarities, or in related problems such as Nearest Neighbor or k-Nearest Neighbor search. Another avenue of research involves scaling up the number of threads and processors used to solve the problem, which in turn will scale the size of the problem that can be efficiently solved.

Acknowledgment: This work was supported in part by NSF (IOS-0820730, IIS-0905220, OCI-1048018, CNS-1162405, and IIS-1247632) and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

References

[1] C. Xiao, W. Wang, X. Lin, and J. X. Yu, "Efficient similarity joins for near duplicate detection," in *Proceedings of the 17th International Conference on World Wide Web*, ser. WWW '08. New York, NY, USA: ACM, 2008, pp. 131–140.

[2] M. Sahami and T. 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*, ser. WWW '06. New York, NY, USA: ACM, 2006, pp. 377–386.

[3] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 131–140.

[4] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig, "Syntactic clustering of the web," in *Selected papers from the sixth international conference on World Wide Web*. Essex, UK: Elsevier Science Publishers Ltd., 1997, pp. 1157–1166.

[5] T. H. Haveliwala, A. Gionis, and P. Indyk, "Scalable techniques for clustering the web," in *In Proc. of the WebDB Workshop*, 2000, pp. 129–134.

[6] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 271–280.

[7] A. Chandel, O. Hassanzadeh, N. Koudas, M. Sadoghi, and D. Srivastava, "Benchmarking declarative approximate selection predicates," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '07. New York, NY, USA: ACM, 2007, pp. 353–364.

[8] D. Lee, J. Park, J. Shim, and S.-g. Lee, "An efficient similarity join algorithm with cosine similarity predicate," in *Proceedings of the 21st International Conference on Database and Expert Systems Applications: Part II*, ser. DEXA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 422–436.

[9] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[10] A. Hotho, A. Nürnberger, and G. Paaß, "A brief survey of text mining," *LDV Forum - GLDV Journal for Computational Linguistics and Language Technology*, 2005.

[11] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *Proceedings of the 22nd International Conference on Data Engineering*, ser. ICDE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 5–.

[12] J. Zhai, Y. Lou, and J. Gehrke, "Atlas: a probabilistic algorithm for high dimensional similarity search," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 997–1008.

[13] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *Proceedings of the 32nd international conference on Very large data bases*, ser. VLDB '06. VLDB Endowment, 2006, pp. 918–929.

[14] C. Xiao, W. Wang, and X. Lin, "Ed-join: an efficient algorithm for similarity joins with edit distance constraints," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 933–944, Aug. 2008.

[15] C. Xiao, W. Wang, X. Lin, and H. Shang, "Top-k set similarity joins," in *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ser. ICDE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 916–927.

[16] L. A. Ribeiro and T. Härder, "Efficient set similarity joins using min-prefixes," in *Proceedings of the 13th East European Conference on Advances in Databases and Information Systems*, ser. ADBIS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 88–102.

[17] J. Wang, G. Li, and J. Feng, "Can we beat the prefix filtering?: an adaptive framework for similarity join and search," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 85–96.

[18] A. Awekar and N. F. Samatova, "Fast matching for all pairs similarity search," in *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology - Volume 01*, ser. WI-IAT '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 295–300.

[19] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, ser. STOC '98. New York, NY, USA: ACM, 1998, pp. 604–613.

[20] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases*, ser. VLDB '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 518–529.

[21] V. Satuluri and S. Parthasarathy, "Bayesian locality sensitive hashing for fast similarity search," *Proc. VLDB Endow.*, vol. 5, no. 5, pp. 430–441, Jan. 2012.

[22] A. Moffat, R. Sacks-davis, R. Wilkinson, and J. Zobel, "Retrieval of partial documents," in *Information Processing and Management*, 1994, pp. 181–190.

[23] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, "Rcv1: A new benchmark collection for text categorization research," *J. Mach. Learn. Res.*, vol. 5, pp. 361–397, Dec. 2004.

[24] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW '10: Proceedings of the 19th international conference on World wide web*. New York, NY, USA: ACM, 2010, pp. 591–600.

[25] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proc. Internet Measurement Conf.*, 2007.