ATLAS: A Probabilistic Algorithm for High Dimensional Similarity Search

Jiaqi Zhai
Cornell University
Ithaca, NY
jz392@cornell.edu

Yin Lou
Cornell University
Ithaca, NY
yinlou@cs.cornell.edu

Johannes Gehrke
Cornell University
Ithaca, NY
johannes@cs.cornell.edu

ABSTRACT

Given a set of high dimensional binary vectors and a similarity function (such as Jaccard and Cosine), we study the problem of finding all pairs of vectors whose similarity exceeds a given threshold. The solution to this problem is a key component in many applications with feature-rich objects, such as text, images, music, videos, or social networks. In particular, there are many important emerging applications that require the use of relatively low similarity thresholds.

We propose ATLAS, a probabilistic similarity search algorithm that in expectation finds a $1-\delta$ fraction of all similar vector pairs. ATLAS uses truly random permutations both to filter candidate pairs of vectors and to estimate the similarity between vectors. At a 97.5% recall rate, ATLAS consistently outperforms all state-of-the-art approaches and achieves a speed-up of up to two orders of magnitude over both exact and approximate algorithms.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process, clustering*

General Terms

Algorithms, Experimentation, Performance

Keywords

Similarity Search, Set Similarity Join, Data Mining

1. INTRODUCTION

Finding similar objects is a key component in many applications, such as document and image clustering [14, 23], plagiarism detection [30], near duplicate documents detection [24], 3D scene reconstruction [5], similar music and video retrieval [19, 32], community mining [31], and personalized recommendations [17].

A natural way to describe complex objects is to represent them as vectors in a high dimensional space, where the dimensions correspond to the features extracted from the objects. For example, there are hundreds of thousands of words in an unabridged English dictionary, and usually each word is considered a feature. Standard content-based image retrieval algorithms preprocess images by extracting local features [26] and quantizing them into visual words [27]; research shows that retrieval systems using a million visual words tend to outperform those using a smaller visual vocabulary [27]. Large-scale recommendation systems need to find the similarities of users over millions of items [17]. In many important applications, these vectors are either binary, or can be approximated by binary vectors. In this paper, we focus on the special case of similarity search on binary vectors where the dimensionality of the vectors is on the order of 10^5 to 10^8 dimensions.

Many emerging applications need to measure similarities and differences between objects with low similarity thresholds. For example, consider the problem of grouping web pages. Haveliwala et al. proposed a method to efficiently cluster 12 million web pages [23] by first representing each web page as a set of words, then finding all pairs of web pages whose similarities are above a specific threshold, and finally applying a link-based clustering algorithm to compute the clusters. The similarity threshold in their application is only 0.2[1]. As a second example, consider the problem addressed by Agarwal et al. that reconstructs 3D scenes from a very large collection of photos by finding geometrically consistent images [5]. Each image is represented by a set of visual words, and similarities between images are computed in order to discard unpromising image pairs. However, most geometrically consistent images have similarity between 0.025 and 0.1! As a third example, consider the problem of finding plagiarism in large document collections. Sorokina et al. designed a mechanism to find plagiarism in research document collections [30]; they represented each document as a set of 7-grams, i.e., subsequences containing seven consecutive words. The authors found that most plagiarism cases only have four sentences in common. As each document contains hundreds of sentences, this corresponds to a very low similarity threshold. However, previous research on high dimensional similarity search has mainly focused on similarity thresholds above 0.5 [7, 8, 35, 33]. These thresholds are primarily useful for tasks such as finding near-duplicate documents and retrieving similar text snippets [35, 28].

Beyond reasons that are motivated by specific applications, the use of low similarity thresholds also has an intu-

SIGMOD'11, June 12–16, 2011, Athens, Greece.

[1]We assume that similarity is normalized in the interval [0,1].

itive justification based on the geometrical properties of a high dimensional space [10]. Consider the volume V_d of a d-dimensional hypersphere with radius R. It can be computed using the following formula:

$$V_d = \frac{\pi^{\frac{d}{2}}}{\Gamma(\frac{d}{2}+1)} R^d,$$

where Γ is the Gamma function. Note that the fraction V_d/R^d goes to zero as d goes to infinity. To put these facts into the context of similarity search, assume that we select two points, A and B, uniformly at random from the unit hypercube. We are interested in the Euclidean distance between A and B. Consider a hypersphere of radius 0.99 centered on A. When $d = 25$, the probability that B is inside the hypersphere is at most 7.45×10^{-4}. However, when $d = 100$, this probability is at most 8.67×10^{-41}, and it is at most 7.45×10^{-110} when $d = 200$. This simple example illustrates a counterintuitive fact: two objects with a very low similarity value could still be very similar since the volume of a d-dimensional hypersphere becomes vanishingly small relative to that of a d-dimensional hypercube as d increases.

Traditional methods, such as kd trees and R-trees, only work well in low dimensions (typically less than one hundred). Recent attempts to speed up similarity search focus on reducing the search space of candidate pairs whose similarity needs to be compared. These algorithms are either exact, in that their outputs contain all pairs of vectors whose similarity is above the threshold, or approximate, in that they only give probabilistic guarantees on their outputs. The fastest exact algorithms, such as All-Pairs [8] and PPJoin [35], use an inverted index and apply various heuristics (such as prefix-filtering or ordering of the dataset) to reduce the size of the search space. These approaches require the user to preprocess the data, and in our experiments, their performance degrades quickly as the size of the index increases. Approximate algorithms have been developed based on locality sensitive hashing (LSH) [25], a probabilistic scheme that hashes similar vectors into the same buckets with high probability. LSH-based algorithms are often much faster than exact algorithms, but they do not work well for low similarity thresholds. Bayardo et al. state that their approach is faster than LSH but did not evaluate Min-Hash [8]. Other experimental evaluations also did not include Min-Hash [35, 33].

In this paper, we propose a novel concept for similarity search that is based on the use of *random filters*. Algorithms built on this idea repeatedly apply random filters to discard irrelevant vector pairs; thus, we call them *filtering-based* algorithms. We show that we can use truly random permutations to cheaply generate each random filter, then use inverted indexes to compute the intersection between the outputs of all filters to arrive at the candidate set. In addition, we demonstrate that truly random permutations can also be used to reliably estimate the similarity between vector pairs. This leads to a new candidate filtering algorithm, which works especially well for objects whose vector representation typically contains many non-zero elements, such as text, images, and videos. Finally, we observe that cluster-like structures abound in real data; text documents can be described with a set of topics, and photographs are often taken near objects of interests. We propose a clustering algorithm that exploits this property for further speed-up.

The main contributions of this paper are as follows:

- We propose a filtering-based algorithm for generating vector pairs whose similarities are likely to be above the given threshold. This algorithm significantly outperforms other candidate generation algorithms for low similarity thresholds, supports incremental queries, and can be easily parallelized.
- We present an efficient algorithm for estimating the similarities between vector pairs. We show analytically that we only need to examine a fixed fraction of the dimensions in order to discard irrelevant vector pairs.
- We present a fast approximate clustering algorithm that exploits cluster structures in real data. Our algorithm uses random sampling and probabilistic assignments, and can reduce the search space by up to 92% while discarding less than 1% of the true positives.
- Based on these three ideas, we develop a novel probabilistic similarity search algorithm called ATLAS, and we provide experimental results on several real-world datasets. At a 97.5% recall rate, ATLAS is up to 210 times faster than previous exact algorithms and up to 80 times faster than previous approximate algorithms.

The rest of the paper is organized as follows. We first give the problem definition and introduce some notation in Section 2. Then, we present our permutation-based algorithms for similarity search in Section 3, and our approximate clustering algorithm in Section 4. We consider how to integrate these algorithms and discuss scalability issues in Section 5. In Section 6, we evaluate our algorithm against current state-of-the-art algorithms. Based on our experimental results, we discuss the strengths and weaknesses of existing similarity search algorithms in Section 7. We discuss related work in Section 8 and conclude in Section 9.

2. PRELIMINARIES

Given a set of n binary vectors $V = \{v_1, v_2, \cdots, v_n\}$ in $\{0,1\}^d$ (we number the d dimensions $1, 2, \ldots, d$), a similarity function $sim(x, y)$, and a similarity threshold t, we wish to find all vector pairs $\{x, y\}$ such that $sim(x, y) > t$. We assume that $sim(x, y)$ is normalized in the interval $[0, 1]$, is symmetric, and monotonically increases with $dot(x, y)$ when we hold the number of non-zero dimensions in both x and y constant. To trade off accuracy for efficiency, we allow the algorithm to only output a $(1-\delta)$ fraction of all such vector pairs in expectation. We call this problem *δ-approximate similarity search*.

For high-dimensional similarity search problems, the vectors in V are usually sparse in that they have few non-zero entries. We represent a sparse vector v_i as a list by enumerating the indices of its non-zero entries in increasing order: $v_i = [v_i^0, v_i^1, v_i^2, \ldots, v_i^{|v_i|-1}]$. We call $|v_i|$ the *length* of v_i, and use L to denote the average length of the vectors in V, $L = \sum_{i=1}^{n} |v_i|/n$.

A permutation $\pi : \{1, 2, \ldots, d\} \rightarrow \{1, 2, \ldots, d\}$ defines a reordering of the d dimensions. We denote the result of applying a permutation π to v_i by $\pi(v_i)$. $\pi(v_i)^j$ is the j-th smallest element in the list v_i under the permutation $(0 \leq j \leq |v_i| - 1)$. We represent the set of v_i's K smallest elements by $F_{v_i}^{\pi,K} = \{\pi(v_i)^0, \pi(v_i)^1, \ldots, \pi(v_i)^{K-1}\}$.

Similarity search algorithms usually consist of two stages: *candidate generation* and *candidate filtering*. In the first

Notation	Description
V	The set of binary vectors
n	$n = \|V\|$, the number of vectors in V
d	The dimensionality of the vectors in V
t	The similarity threshold
sim	The similarity function
δ	The error rate
π	A permutation of $\{1, 2, \ldots, d\}$
x	A vector in V
$\|x\|$	The number of non-zero entries in x
L	The average length of vectors in V
C^V	Candidate pairs of vectors from V

Table 1: Notation Summary

stage, the algorithm selects candidate pairs of vectors from V that are likely to have similarity above t. In the second stage, the algorithm verifies if the similarity of each candidate pair is indeed above t. We follow that general framework in this paper. We use C^V to denote the set of unordered candidate pairs, and C^V_x to denote $\{y \mid \{x, y\} \in C^V\}$.

Our algorithm may further split V into several subsets, $V_1, V_2, \ldots, V_m$. When this is the case, we use C^{V_i} to denote the set of candidate pairs in V_i, and $C^{V_i}_x$ to denote the set of candidate vectors for x in V_i, respectively.

We focus on the two most commonly used similarity functions, Jaccard and Cosine, which are defined as follows:

$$\begin{aligned} \text{Jaccard}(x, y) &= dot(x, y) \;/\; (|x| + |y| - dot(x, y)) \\ \text{Cosine}(x, y) &= dot(x, y) \;/\; \sqrt{|x| \cdot |y|} \end{aligned}$$

In Table 1, we summarize the most important notation.

3. PERMUTATION-BASED ALGORITHMS

Many similarity functions are computed based on the number of non-zero elements shared by the two vectors, x and y. Recall that in our problem setting, $sim(x, y)$ increases with $dot(x, y)$ when we hold both $|x|$ and $|y|$ constant. For Jaccard similarity,

$$\frac{dot(x, y)}{|x| + |y| - dot(x, y)} > t \Leftrightarrow dot(x, y) > \frac{t}{1 + t} (|x| + |y|)\,; \tag{1}$$

and for Cosine similarity,

$$dot(x, y)/\sqrt{|x| \cdot |y|} > t \Leftrightarrow dot(x, y) > t\sqrt{|x| \cdot |y|}. \tag{2}$$

We can use Equations (1) and (2) by applying a truly random permutation to the d dimensions. This allows us to develop efficient methods for the *candidate generation* step and for the *candidate filtering* step in a similarity search algorithm. We define the following subproblems:

PROBLEM 1. *(Candidate Generation) Given V, sim, and t, return C^V such that $(1 - \delta_1^*)$ fraction of all vector pairs $\{x, y\}$ with $sim(x, y) > t$ are in C^V.*

PROBLEM 2. *(Candidate Filtering) Given V, sim, t, and C^V, determine whether each vector pair $\{x, y\} \in C^V$ is likely to satisfy $sim(x, y) > t$ in time $O(|V|L) + o(|C^V|L)$ with $(1 - \delta_2^*)$ probability of success.*

and present our solutions in Sections 3.1 and 3.2.

3.1 Candidate Generation

To illustrate the intuition behind our filtering-based candidate generation algorithm, assume that $A_1, A_2, \ldots, A_n$ is a list of random variables such that each one of them can only take one of two values, 0 and 1, and that exactly m among them have value 1. Because these m non-zero elements divide the list into $m+1$ blocks, the expected position of the first non-zero element A^* among them, P_{A^*}, should be approximately $n/(m+1)$. Furthermore, P_{A^*} is extremely unlikely to be very large. For $n = 1000$ and $m = 100$, the probability that P_{A^*} is larger than 50 is less than 0.005.

This motivates us to use a simple procedure for candidate generation: apply a random permutation to the d dimensions, build an inverted index that contains each vector's smallest K non-zero elements, and finally look up the inverted index to find the candidate pairs. Here, K is chosen such that for two vectors x and y, if $sim(x, y) > t$, then with high probability, the intersection of their first K non-zero dimensions is non-empty.

EXAMPLE 1. *Consider four vectors: $a = [1, 3, 5, 7]$, $b = [2, 3, 5, 7]$, $c = [2, 3, 4, 6]$, and $d = [1, 2, 4]$, and a permutation $\pi_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 4 & 2 & 6 & 5 & 7 & 3 & 1 \end{pmatrix}$. Under this permutation, $\pi_1(a) = [7, 1, 3, 5]$, $\pi_1(b) = [7, 2, 3, 5]$, $\pi_1(c) = [2, 6, 4, 3]$, and $\pi_1(d) = [2, 1, 4]$. For $K = 1$, this procedure will index the element 7 in a, the element 7 in b, the element 2 in c, and the element 2 in d. The candidate set is $C_1 = \{\{a, b\}, \{c, d\}\}$. If $K = 2$, then the candidate set will be $C_2 = \{\{a, b\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}\}$ instead.*

This procedure only needs to index a small fraction of the non-zero elements in each vector even when t is low, and thus it is extremely fast. The candidate filtering step may become the bottleneck in the algorithm if we have too many candidates. We can further reduce the size of the candidate set (and also the time needed for the candidate filtering step) by running the above procedure S times and taking the intersection of all candidate sets. Each run of the above procedure can be regarded as the use of a *random filter*, and each candidate set as the output of a random filter. The algorithm works because pairs of vectors with high similarity scores are more likely to be included in the output than pairs with low similarity scores.

EXAMPLE 2. *Consider again the four vectors from Example 1. We apply $\pi_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 5 & 3 & 1 & 4 & 2 & 7 & 6 \end{pmatrix}$, and again index the first two non-zero elements in each vector. The candidate set is then $C_2' = \{\{a, b\}, \{a, c\}, \{b, c\}, \{c, d\}\}$. We observe that $C_2' \cap C_2 = \{\{a, b\}, \{b, c\}, \{c, d\}\}$, which happens to contain exactly all vector pairs that have at least two elements in common.*

If we repeat this procedure S times, the probability that a pair of vectors $\{x, y\}$ is present in the final candidate set C^V is

$$\mathbf{Pr}\left[|F_x^{\pi_s, K} \cap F_y^{\pi_s, K}| > 0, \forall s\right] = \left(1 - \frac{\binom{|x \cup y| - |x \cap y|}{K}}{\binom{|x \cup y|}{K}}\right)^S. \tag{3}$$

Our algorithm can be implemented with S inverted indexes. The s-th inverted index, I_s, consists of d lists, $I_s^1, I_s^2, \ldots, I_s^d$. We give the pseudo code in Algorithm 1.

Algorithm 1 ATLAS: Candidate Generation

Input: V; S; K
Output: C^V

```
C^V ← all possible vector pairs in V
for s = 1 to S do
    π_s ← a random permutation of the d dimensions
    ∀ x ∈ V, apply π_s to get F_x^{π_s,K}
    I_s ← an inverted index built from all F_x^{π_s,K} where (x ∈ V)
    C' ← all {x, y} that share at least one indexed feature
    C^V ← C^V ∩ C'
end for
return C^V
```

A naive implementation of Algorithm 1 uses $O(n^2)$ memory. We can avoid this by computing the candidate set, C_x^V, for each x separately. We give the pseudo code in Algorithm 2. This procedure only requires linear space and can be easily parallelized.

Algorithm 2 Finding Candidate Vectors For Vector x

Input: $x \in V$; $I_1, I_2, \ldots, I_S$; $F_x^{\pi_1,K}, F_x^{\pi_2,K}, \ldots, F_x^{\pi_S,K}$
Output: C_x^V

```
C_x^V ← {1, 2, ..., |V|}
for s = 1 to S do
    C'_x ← ∅
    for all p ∈ F_x^{π_s,K} do
        C'_x ← C'_x ∪ I_s^p
    end for
    C_x^V ← C_x ∩ C'_x
end for
return C_x^V
```

Note that the probability of $\{x, y\} \in C^V$ is closely related to the ratio of $|x \cap y|$ to $|x \cup y|$. For Cosine similarity, this probability depends on both $|x|$ and $|y|$, so we cannot directly bound the recall rate. Nevertheless, in many practical applications, the lengths of the vectors do not vary too much. In such cases, we can use this algorithm for Cosine similarity at a cost of longer running times.

The space complexity of the algorithm is $O(nKS+d)$. The main overhead comes from the space needed to store the S inverted indexes. The time complexity is $O(n^2S + nS(L + K)+Sd+nL\log L)$. Here, $O(nS(L+K))$ is the time needed for generating the inverted indexes, and $O(n^2S)$ is the time needed for generating the candidate pairs. In practice, the number of vector pairs with similarity above t is on the order of $o(n^2)$, so for suitable parameters, the time needed for candidate generation will never reach $O(n^2S)$. In our experiments, we find that we can obtain a good estimation of the parameter values via random sampling; we discuss our sampling procedure in Section 5.1.

3.1.1 Comparison with Existing Candidate Generation Algorithms

The efficiency of a similarity search algorithm is mostly determined by its candidate generation algorithm. Existing candidate generation algorithms can be categorized into two groups: *signature-based schemes*, and *inverted index-based schemes*. A signature-based algorithm first generates a set of signatures for each vector. Then, two vectors are considered as a candidate if they share at least one common signature. Algorithms that use this idea include Min-Hash [11], random projections-based LSH [12], and PartEnum [7]. In contrast, an inverted index-based algorithm first selects some elements from each vector to index, then computes the candidate set while traversing the inverted index. The fastest algorithms among them are All-Pairs [8] and PPJoin [35].

Our approach is best viewed as a *filtering-based scheme*, because our core idea is to repeatedly apply a series of random filters to the $O(n^2)$ vector pairs. To efficiently apply these filters, we combine insights from both the signature-based algorithms and the inverted index-based algorithms. Our selection of the first K non-zero dimensions in each vector can be viewed as generating K signatures for each vector. Then, we use inverted indexes to efficiently compute the intersection of candidate sets.

Comparison with Min-Hash: Another probabilistic similarity search algorithm, Min-Hash, also uses random permutations [11]. Min-Hash is a LSH-based algorithm. Its hash function hashes two vectors into the same bucket with a probability that increases with their Jaccard similarity. If π is a min-wise independent permutation, then

$$\mathbf{Pr}[\pi(x)^0 = \pi(y)^0] = \frac{|x \cap y|}{|x \cup y|} = \text{Jaccard}(x, y). \quad (4)$$

The minimum non-zero dimension under π is called a *token*. A *signature* (or hash key) is formed by combining R tokens. The candidate set is then the vector pairs sharing the same signatures. This process is repeated J times in order to achieve a high recall rate. The resulting probability that we keep $\{x, y\}$ is

$$\mathbf{Pr}\left[\bigcup_{j=1}^{J}\left(\bigcap_{r=1}^{R}\pi_r^j(x)^0 = \pi_r^j(y)^0\right)\right] = 1-\left(1-\left(\frac{|x\cap y|}{|x\cup y|}\right)^R\right)^J. \quad (5)$$

Note that each π_r^j in (5) only needs to be a min-wise independent permutation. In our algorithm, for (3) to hold, we need each π_s to be a K min-wise independent permutation. Directly evaluating a K min-wise independent hash function can be very costly for large Ks; however, in practical applications, d is usually far less than 10^9. This allows us to use a random permutation in order to avoid calculating K min-wise independent hash functions.

We now discuss the performance differences between Min-Hash and our candidate generation algorithm. Note that

$$\left(1-\frac{\binom{|x\cup y|-|x\cap y|}{K}}{\binom{|x\cup y|}{K}}\right)^S \le \left(1-(1-t)^K\right)^S,$$

and the differences between the left-hand side and the right-hand side of the above equation is negligible when K is small relative to the lengths of the vectors. We select K, S, J, and R such that both ATLAS's candidate generation algorithm and Min-Hash have an expected recall rate of $1 - \delta_1^*$ for all pairs of vectors $\{x, y\}$ with $\text{Jaccard}(x, y) = t$:

$$(1-(1-t)^K)^S = 1-(1-t^R)^J$$

We take logs on both sides of the above equation. Because δ_1^* is usually very small in practice, we can apply the approximation $\ln(1 + x) \approx x$ for $|x| < 1$:

$$S(1-t)^K = (1-t^R)^J.$$

It directly follows that

$$K = \frac{\ln(1-t^R)}{\ln(1-t)} J - \frac{\ln S}{\ln(1-t)}. \quad (6)$$

We immediately observe that $r = \ln(1-t^R)/\ln(1-t)$ goes to 0 as t approaches 0. For $R = 3$, $r \approx 0.078$ when $t = 0.3$, 0.0095 when $t = 0.1$, and 0.00062 when $t = 0.025$, and r becomes even smaller as R increases. Since the efficiency of the overall algorithm is strongly related to the number of false positive candidates, and the probability of generating false positives is controlled by R in Min-Hash and S in ATLAS, we can interpret Equation (6) to mean that Min-Hash requires a significant increase in J to compensate for a small increase in R. This suggests that ATLAS is more efficient than Min-Hash for lower similarity thresholds. In Section 6, we experimentally show that ATLAS is generally faster than Min-Hash when $t \leq 0.5$.

We show these effects in a simulation-based analysis that uses datasets from Section 6. Consider the probability of selecting a pair $\{x, y\}$ as a candidate. We set the expected value of $\mathbf{Pr}[\{x, y\} \in C^V]$ when $sim(x, y) = t$ to 0.975, and plot the graphs for $t = 0.05$, $t = 0.1$, and $t = 0.2$ in Figure 1, with the parameters that minimize the running times of the algorithms.

Note that Min-Hash generates fewer false positive candidate pairs than ATLAS does when the similarity is close to t, but many more false positives when the similarity is very low. This is because the S in the optimal set of parameters for ATLAS is always higher than the R for Min-Hash. While ATLAS's candidate generation algorithm tends to generate more vector pairs with similarity near t, the number of such vector pairs is, in general, much smaller than the number of vector pairs with similarity significantly less than t. From the graphs, we can also see why the speed-up of ATLAS relative to Min-Hash decreases as t increases.

3.2 Candidate Filtering

Given the candidate vector pairs from the output of the candidate generation algorithm, we are interested in finding a fast and reliable way to discard vector pairs with low similarity values. This is part of the second stage in a typical similarity search algorithm: verifying if the similarity of each candidate pair is indeed above the threshold.

To solve this problem, we consider all $n(n-1)/2$ potential vector pairs as a whole. We can often divide these vector pairs into two sets, A and B, such that the similarity between any vector pairs in A is at least $t^* = \gamma t$ $(0 < \gamma < 1)$ and B contains the remaining vector pairs. Intuitively, t^* is a threshold between noise and non-noise; thus, we expect $|B|$ to be much greater than $|A|$. We could greatly reduce the running time of a similarity search algorithm by reliably discarding vector pairs in B. We introduce a concept, *Threshold Sensitive Projection (TSP)*, to capture this intuition.

Definition 1. A TSP family $\mathcal{F}$ is defined for a vector space $\mathcal{V}$, a similarity function sim, two similarity thresholds $t > t^* > 0$, a probability of error $\epsilon > 0$, an approximation factor $c > 0$, and a threshold $p^* \in \mathbb{R}$. Here, $\mathcal{F}$ is a family of functions $h : (\mathcal{V}, \mathcal{V}) \to \mathbb{R}$ such that for any two points $x, y \in \mathcal{V}$, and a function h chosen uniformly at random from $\mathcal{F}$:

1. If $sim(x, y) > t$, then $\mathbf{Pr}[h(x, y) \in [p^*, \infty)] \geq 1 - \epsilon$,
2. If $sim(x, y) < t^*$, then $\mathbf{Pr}[h(x, y) \in [p^*, \infty)] \leq c\epsilon$.

Essentially, at p^* a *phase transition* occurs; $\{x, y\}$ goes from almost surely being noise (i.e., $sim(x, y) < t^*$), to almost surely being interesting (i.e., $sim(x, y) > t$).

The above definition is similar to the definition of LSH families, but a LSH family is defined for individual vectors, whereas a TSP family is defined for pairs of vectors.

A TSP family is useful only if we can quickly evaluate h. We formalize this in the following definition:

Definition 2. A family of TSP functions $\mathcal{F}$ is *interesting* if for n vectors whose average length is L, the expected time of evaluating $h(x, y)$ for m pairs of vectors among them is $O(nL) + o(mL)$.

Essentially, this definition says that evaluating h for all vector pairs in the candidate set should be asymptotically faster than performing pair-wise comparisons. It is important to note that m is generally much larger than n.

One TSP family can be trivially derived from locality sensitive hashing. In this TSP family, h is just the number of common elements between the signatures of two vectors formed out of R LSH tokens. When the threshold t is low, this approach either takes too much time or does not give us enough discriminative power. Another TSP family is only applicable to binary vectors, but offers better discriminative power. This is the TSP family that we use in our algorithm, and we discuss it in the following subsection.

3.2.1 A TSP Family for Binary Vectors

Consider two binary vectors, x and y. If $sim(x, y) > t$, then we can use Equations (1) and (2) to lower bound their dot product, $dot(x, y)$, by $o = pd$ $(0 \leq p \leq 1)$; on the other hand, if $sim(x, y) < t^*$, then we could upper bound $dot(x, y)$ by $\delta o = \delta pd$ $(0 \leq \delta < 1)$. Here, p and δ are two parameters whose values are determined by the similarity function sim and the two similarity thresholds, t and t^*.

After applying a random permutation π, we check the first αd dimensions of $\pi(x)$ and $\pi(y)$, and we discard $\{x, y\}$ if the number of common non-zero dimensions is less than $\beta o = \beta pd$ $(\alpha > \beta)$. Note that we have

$$\begin{aligned} \mathbf{Pr}[\text{Keep} \mid sim(x, y) > t] &= 1 - H_L(pd, d, \alpha d, \beta pd), \\ \mathbf{Pr}[\text{Discard} \mid sim(x, y) < t^*] &= 1 - H_H(\delta pd, d, \alpha d, \beta pd). \end{aligned}$$

Here, we use $H_L(M, N, n, k)$ and $H_H(M, N, n, k)$ to denote $\mathbf{Pr}[X \leq k]$ and $\mathbf{Pr}[X \geq k]$ for a random variable X taken from a hypergeometric distribution with parameters N, M, and n, where

$$\begin{aligned} \mathbf{Pr}[X = k] = h(M, N, n, k) &= \binom{M}{k}\binom{N-M}{n-k}\binom{N}{n}^{-1}, \\ H_L(M, N, n, k) &= \sum_{i=0}^{k} h(M, N, n, i), \text{and} \\ H_H(M, N, n, k) &= \sum_{i=k}^{n} h(M, N, n, i). \end{aligned}$$

We require $\alpha > \beta$ and $\delta\alpha < \beta$ for the bounds to be meaningful, and we will assume so in the remainder of the paper.

LEMMA 1. *(A modification of Chvátal's bound on the tail probability of hypergeometric distributions)*

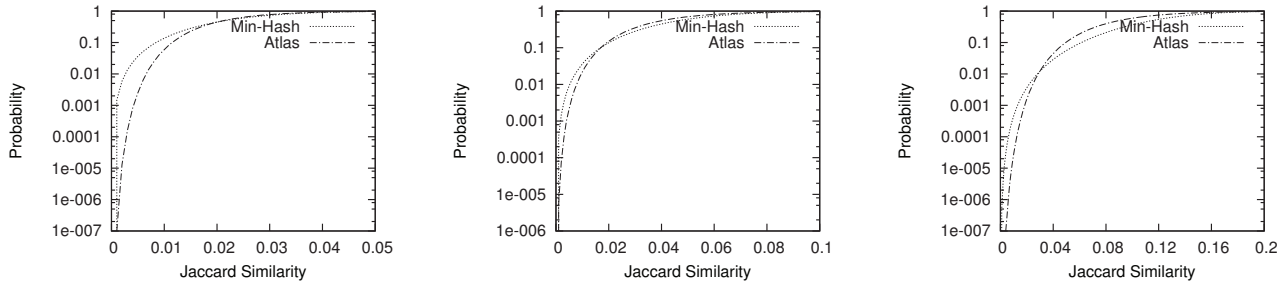


Figure 1: Probability of $\{x, y\} \in C^V$

$$H_H(M, N, n, k) \leq \left(\left(\frac{p}{p+t}\right)^{p+t}\left(\frac{1-p}{1-p-t}\right)^{1-p-t}\right)^n \quad (7)$$

$$\leq e^{-2t^2 n} \quad (8)$$

for $p = M/N$ and $t = k/n - p$ $(t \geq 0)$;

$$H_L(M, N, n, k) \leq \left(\left(\frac{p}{p+t}\right)^{p+t}\left(\frac{1-p}{1-p-t}\right)^{1-p-t}\right)^n \quad (9)$$

$$\leq e^{-2t^2 n} \quad (10)$$

for $p = (N - M)/N$ and $t = (n - k)/n - p$ $(t \geq 0)$.

Proof. The bound for $H_H(M, N, n, k)$ directly follows from [15]. Using the symmetric relation $h(M, N, n, k) = h(N - M, N, n, n - k)$, we obtain the bound for the lower tail of hypergeometric distribution. □

We can now prove the main theorem using Lemma 1.

Theorem 1. *Given pd, the lower bound on the number of non-zero dimensions for vectors with similarity t, and δpd, the upper bound on the number of non-zero dimensions for vectors with similarity t^* $(t^* < t)$, one can decide whether the similarity between the two, $sim(x, y)$, is above t or is below t^* by examining a fixed proportion of the d dimensions, $\alpha = O(\frac{-\ln \epsilon}{p^2(1-\delta)^2 d})$ with probability of success $1 - \epsilon$.*

Theorem 1 states that as the average length of vectors increases, pd also increases, so the absolute number of dimensions that we need to examine *decreases*.

We now show experimentally that the actual value we need for α is quite small. We set d to 1,000,000, the probability of discarding $\{x, y\}$ with similarity t to 0.01, and the probability of keeping $\{x, y\}$ with similarity t^* to 0.05. In Figure 2, we first plot the values of α while varying the expected number of common dimensions, pd, then plot the values of α while varying δ.

In practice, Chvátal's stronger bounds ((7) and (9)) give us a good approximation of the true tail probability. It is best to choose multiple values of δ and compute the corresponding value of α and β based on the data distribution. We show our algorithm for choosing parameters in Section 5.1.

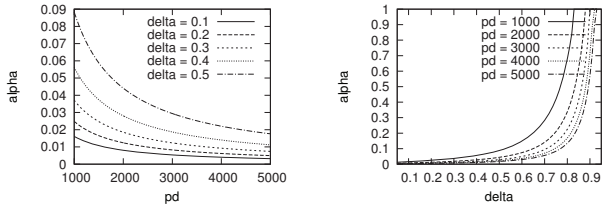


Figure 2: Relationships Between pd, α, and δ

4. VECTOR PRE-CLUSTERING

Vectors in real datasets possess cluster-like structures. Text documents can be described using a set of topics. The majority of photographs are taken near objects of interests, such as landmarks. In social networks and recommendation systems, we can categorize the users into various communities. In these cases, two vectors are likely to be similar if and only if they share at least one common 'topic'. To exploit this property, we need an algorithm that reports if cluster-like structures exist, and if so, assigns vectors to clusters such that we only need to search for similar vectors within each cluster. We define the following subproblem:

Problem 3. *(Vector Pre-Clustering) Given V, sim, and t, find m (possibly overlapping) subsets of V, $V_1, V_2, \ldots, V_m$, such that $(1-\delta_3^*)$ of all vector pairs $\{x, y\}$ with $sim(x, y) > t$ are contained in at least one of the m subsets.*

and present our solution in the following subsection.

4.1 A Pre-Clustering Algorithm

Traditional clustering algorithms assume that the similarities between objects have been pre-calculated and are given to the clustering algorithms as an input. Without such information, we instead assume that the vectors can be divided into m *dense regions*. These dense regions are defined such that for x and y taken from two different regions, the probability that $sim(x, y) > t$ is close to zero. When the size of the smallest dense region is sufficiently large, Guha et al.'s result allows us to identify these regions by sampling:

Theorem 2. *([22]) For a dense region u, if the sample size s satisfies*

$$s \geq fn + \frac{n}{|u|}\log(\frac{1}{\delta}) + \frac{n}{|u|}\sqrt{(\log(\frac{1}{\delta}))^2 + 2f|u|\log(\frac{1}{\delta})}$$

then the probability that the sample contains fewer than $f|u|$ points belonging to u is less than δ, $0 \leq \delta \leq 1$.

We can bound the probability of selecting less than $f|u|$ points from any of the m regions by $m\delta$.

It remains then to find a succinct representation for the dense regions and to identify which points belong to which regions. Common clustering approaches like k-means tend to output spherical shaped clusters with similar radiuses. Real datasets seldom exhibit such properties; our experiments confirmed that such approaches perform poorly. Instead, we note that the density of points in a region tends to decrease as the distance from the center increases. This motivates us to apply a clustering algorithm, Affinity Propagation [20], to find these regions.

Having obtained the region centers, we assign each point to its top T closest regions (as measured by the distance to the region center). When T equals 1, our pre-clustering scheme is roughly the same as a Voronoi decomposition, since each resulting region corresponds to a high dimensional Voronoi cell. We need to increase T for the cases in which a pair of points are contained in two different Voronoi cells but are close to their common boundary. In our experiments, setting T to 3 or 4 usually gives excellent results.

Finally, we need a way to verify if the resulting clusters are accurate enough for similarity search. We can estimate this by sampling a small number of the vector pairs with similarity above t and calculating the proportion among them that are covered by the clusters. We use a LSH-based algorithm to efficiently generate these samples. For Jaccard similarity, we use Min-Hash as specified by the equation (4). For Cosine similarity, we use random projections-based LSH which is given by the equation

$$\mathbf{Pr}\left[sgn(x \cdot r) = sgn(y \cdot r)\right] = 1 - \frac{\theta(x,y)}{\pi} \sim \text{Cosine}(x,y) \tag{11}$$

where r is a random point on the unit hypersphere [21], $\theta(x,y)$ is the angle between the two vectors, and $sgn(x)$ takes 1 when $x \geq 0$ and -1 otherwise. Since LSH-based algorithms are more prone to generate vector pairs with higher similarities, we need to adjust the weights placed on the samples. In practice, a sample size of 500 usually provides a good estimation.

Algorithm 3 describes an outline of our method. The algorithm consists of three stages: (1) random sampling, (2) clustering, and (3) vector labeling and verification. Note that if our proposed clustering procedure fails to work, Algorithm 3 will simply return a single cluster containing all n points.

Algorithm 3 ATLAS: Vector Pre-Clustering

Input: V; δ_3^*; number of samples r; estimated upper bound M_{max} and lower bound M_{min} on the number of clusters
Output: $(V_1, V_2, \ldots, V_m)$

$V' \leftarrow$ a random sample of size r chosen from V
repeat
  adjust the parameters of the clustering algorithm
  $(V_1', V_2', \ldots, V_m') \leftarrow$ run affinity propagation algorithm on V' (see [20] for details).
until m is between M_{min} and M_{max}.
$\forall\, x \in V, \forall\, 1 \leq i \leq m$, calculate $sim(x, center(V_i'))$
Find sample pairs with $sim(x,y) > t$ and their weights
$T \leftarrow 1$
repeat
  assign each point in V to its closest T regions in order to obtain $(V_1, V_2, \ldots, V_m)$
  if the estimated recall rate is above $1 - \delta_3^*$ **then**
    return $(V_1, V_2, \ldots, V_m)$
  end if
  $T \leftarrow T + 1$
until T or the size of an element $V_i \in (V_1, V_2, \ldots, V_m)$ is sufficiently large
return (V)

Algorithm 4 ATLAS: Framework

Pre-cluster V into $(V_1, V_2, \ldots, V_m)$.
$O \leftarrow \emptyset$
for $i = 1$ to m **do**
  $C^{V_i} \leftarrow$ candidate vector pairs in V_i
  for all $\{x,y\} \in C^{V_i}$ **do**
    if $\{x,y\}$ passes the filtering test and $sim(x,y) > t$ **then**
      $O \leftarrow O \cup \{\{x,y\}\}$
    end if
  end for
end for
return O

5. INTEGRATION AND SCALABILITY

We can integrate the three algorithms that we have previously discussed in Sections 3 and 4 as shown in Algorithm 4. While we need to find suitable values of δ^* for each of the three subproblems in order to integrate them, we find that the algorithm is not extremely sensitive to the values used, and a relatively rough estimation tends to work well in practice.

Note that we can use the vector pre-clustering result in two ways. The first way is the procedure outlined in Algorithm 4. This is generally efficient, but cannot be easily parallelized since it may be difficult to balance the workload across machines. The second way is to check the vector pre-clustering result after a candidate pair is generated, and discard it if all clusters that the two vectors belong to are different.

5.1 Parameter Estimation

5.1.1 Candidate Generation

In this section, we discuss how to find the optimal values of S and K that both minimize the running time of the algorithm and ensure an expected recall rate of $1 - \delta_1^*$. Our conclusion can be applied both to our algorithm, ATLAS, and to locality sensitive hashing-based algorithms, such as Min-Hash.

We cannot obtain good values for S and K without having some prior knowledge about the data distribution. (3) only shows what S and Ks are necessary for the expected recall rate for a single vector pair $\{x,y\}$ to be $1 - \delta_1^*$ when Jaccard(x,y) is t. It is impossible to directly estimate the recall rate for Cosine similarity.

In our experiments, we estimate S and K by running the algorithms on a small sample dataset. We find that the parameters on a sample set of size between 10,000 and 20,000 lead to similar recall rates when we apply them on the larger datasets consisting of millions of records. These parameters,

however, may not minimize the algorithm's running times on the larger datasets. We might need to increase S slightly to reduce the number of candidates for a larger dataset.

5.1.2 Candidate Filtering

As noted in Section 3.2, it is usually best to select multiple (α, β) pairs for our candidate filtering algorithm. To generate these pairs, we assume that we have a list of δs, $\delta_1, \delta_2, \ldots, \delta_t$, that can be used to separate t and different t^*s, and we would like to select P among them for generating the (α, β) pairs. When we know nothing about the data, we can simply use a list like $\delta_1 = 0.05, \delta_2 = 0.10, \ldots, \delta_{19} = 0.95$. We use $P = 3$ for all our experiments. This value can be increased or decreased depending on the magnitude of pd.

We give an outline of the procedure we used for calculating these parameters in Algorithm 5. We assume that the user will give us an upper bound on the probability of discarding vector pairs with similarity above t, p_d, and a lower bound on the probability of keeping vector pairs with similarity below t^*, p_k. The last step in the algorithm selects (α_i, β_i)s with an objective of avoiding applying this filtering procedure too frequently. Other heuristics can be used to replace this step for different datasets.

Algorithm 5 Selecting Parameters for ATLAS's Candidate Filtering Algorithm (Threshold Sensitive Projection)

Input: d; $\delta_1, \delta_2, \ldots, \delta_t$ $(\delta_1 < \delta_2 < \ldots < \delta_t)$; pd; p_d; p_k; P $(P \geq t)$.
Output: $(\alpha_1, \beta_1), (\alpha_2, \beta_2), \ldots, (\alpha_P, \beta_P)$

```
for i = 1 to P do
  α_min ← 0, α_max ← 1.
  while α_min + ε < α_max do
    α_cur ← (α_min + α_max)/2
    β_cur ← the maximum β ∈ (α_cur · δ_i, α_cur) that satisfies Pr[Discard | dot(x,y) ≥ pd] ≤ p_d for α_cur.
    if Pr[Keep | dot(x,y) ≤ δ_i pd, α_cur, β_cur] ≤ p_k then
      α_min ← α_cur
      α_i ← α_cur, β_i ← β_cur
    else
      α_max ← α_cur
    end if
  end while
end for
Select P (α_i, β_i) pairs, (α_{s_1}, β_{s_1}), ..., (α_{s_P}, β_{s_P}), such that min_{1≤i≤P−1} (α_{s_{i+1}} − α_{s_i}) is maximized
```

5.2 Scalability

For large datasets containing millions of vectors, a similarity search algorithm needs to address two issues: (1) how to avoid the memory bottleneck, and (2) how to efficiently use multiple cores on multiple machines.

Most similarity search algorithms need to store all vectors in memory. If storing each non-zero dimension takes 4 bytes, and the average length of the vectors is 2,000, then we need 8 Gbytes to store 1,000,000 binary vectors. This does not account for the memory needed for the additional data structures.

A parallel algorithm can partially address this issue by block-based processing. When we have m machines, we can divide V into m disjoint blocks of equal size, $V_1, V_2, \ldots, V_m$. Then, we assign the i-th block and half of the remaining vectors, V_i^C, to machine i. Machine i first performs a similarity search on V_i, then finds vector pairs such that one of them is in V_i and the other is in V_i^C. We can select V_i^C such that each vector pair is only processed once. This parallelization scheme does not perform any redundant computations, but requires the algorithm to be incremental. We use this parallelization scheme for evaluating ATLAS and Min-Hash in Section 6.4.

Note that a consequence of Equation (6) from Section 3.1.1 is that for low similarity thresholds, Min-Hash needs to use a significant amount of memory in order to support incremental queries.

Dataset	n	d	L
London	903,833	999,935	5,909
ClueWeb	2,255,129	4,238,871	796
arXiv	286,488	800,102	761
DBLP	31,700,425	461,818	15

Table 2: Statistics of the Datasets

6. EVALUATION

We evaluated our algorithm against previous exact and approximate algorithms on four datasets with different characteristics. We tuned the parameters of all approximate algorithms to have a 97.5% recall rate. To understand the performance and scalability of the algorithms, we first evaluated their single-core performance on the smaller datasets, then compared their parallel performance on the larger datasets. We performed the parallelized experiments on a cluster of 51 machines, one of which is used for task scheduling. Each machine is equipped with two 2.66GHz Quad Core Intel Xeon Processors and 16 Gbytes of memory. We used a machine with identical hardware for single-core experiments.

6.1 Datasets

London. This dataset consists of images collected from Flickr [1] by searching for the keyword "London". We first extracted SIFT features from the images [26], then we used a vocabulary tree-based approach [27] to learn one million visual words. Each image is represented as a bag of visual words.

ClueWeb. This dataset consists of web pages from the TREC Category B dataset [2]. For each web page, we applied Krovetz's stemming algorithm and filtered common stop words. We then discarded web pages with less than 500 words.

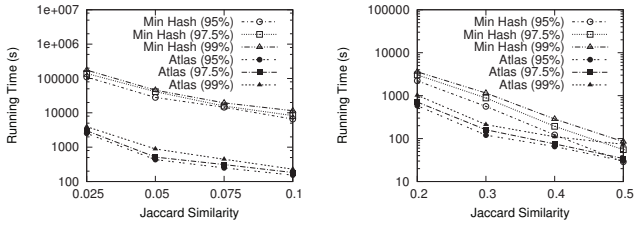
arXiv. This dataset contains a collection of arXiv [3] research articles from 1991 through early 2005. It was first used in by Sorikina et al. for plagiarism detection [30]. We did not apply stemming and stop word filtering on this dataset.

DBLP. This dataset contains a snapshot of the DBLP publication records [4], and has been previously used in the context of set-similarity join [33]; we used their procedure to increase the size of this dataset by a factor of 25 times [33].

Statistics about the datasets are included in Table 2. We did not consider the DBLP dataset suitable for similarity search, as each record in it is simply the title and the list of authors of a paper. However, the data duplication procedure used by Vernica et al. introduces natural cluster-like struc-

Dataset		Exact	Approximate
ClueWeb100K	Jaccard	29.1x–67.2x	0.6x–4.5x
	Cosine	4.8x–91.7x	1.2x–2.7x
London100K	Jaccard	15.9x–210.2x	44.6x–79.3x
	Cosine	1.1x–95.3x	6.4x–80.9x

Table 3: Relative Speed-Up



(a) London100K (b) ClueWeb100K

Figure 4: Recall Rate

tures, making DBLP a good choice for testing our vector pre-clustering algorithm [33]. Since the speed of the exact algorithms are very slow, we created two additional datasets each consisting of 100,000 randomly sampled records, which are called ClueWeb100K and London100K.

6.2 Permutation-Based Algorithms

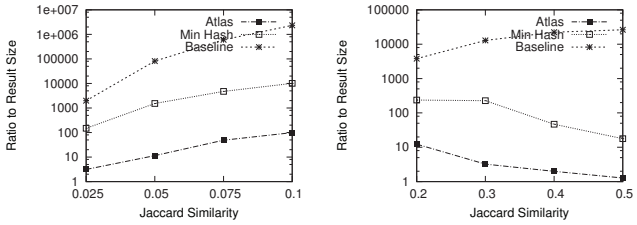
We compared ATLAS's candidate generation and candidate filtering algorithms with All-Pairs [8], PPJoin [35], and Min-Hash [11]. We show the results in Figure 3 (a), (b), (e) and (f), and summarize the comparison in Table 3. For low similarity thresholds, ATLAS's speed-up over Min-Hash generally increases as t decreases. As the number of vector pairs with similarity above t increases, the relative speed-up of both approximate algorithms over exact algorithms decreases.

We are also interested in the impact of recall rates on the approximate algorithms. As we have shown in Figure 4, recall rates between 95% and 99% have relatively little effects on the running time of the algorithms.

Both All-Pairs and PPJoin work poorly for low similarity thresholds. All-Pairs maintains an inverted index dynamically to exploit t. It first sorts the vectors in increasing order of their size and the dimensions in decreasing order of their frequency. Then, it processes the vectors sequentially and only indexes the first $(1-t)|x|$ non-zero dimensions in a vector x. PPJoin contains several improvements to All-Pairs. Its main difference from All-Pairs is that it sorts the dimensions in the increasing order of their frequency, and applies a heuristic, *positional filtering*, that further reduces the number of candidates. When t is low, these algorithms are slow as they need to index almost all the non-zero dimensions.

We also experimented with PPJoin+, an improved version of PPJoin [35]. Our results show that PPJoin+ is consistently slower than PPJoin when L is large, because PPJoin+ employs an additional suffix filtering algorithm, which uses an estimation of the Hamming distance between two set of tokens for pruning. Such a procedure is costly and works well only when it can give a reasonably accurate estimation at a small cost; however, this happens only if L is small.

We have already described a LSH-based algorithm, Min-Hash, in Section 3.1.1. Min-Hash does not work well for low



(a) London100K (b) ClueWeb100K

Figure 5: Candidate Size

similarity thresholds. Consider the case when we use the Jaccard similarity and $t = 0.025$. For $R = 1$, we only need J to be 146 to ensure that each vector pair with similarity t has at least 97.5% probability of being considered; however, this set of parameters would result in many false positive candidates. For $R = 2$, we need to choose $J = 4700$ to have the same recall rate. In this case, the algorithm will spend a long time building signatures and thus candidate generation will be very slow. Thus, it is not possible to find a good tradeoff between the time needed to build signatures and the time needed to filter candidates. The parameters chosen by our algorithm, in contrast, are $S = 8$ and $K = 130$.

An alternative LSH-based algorithm, which we have omitted from our comparisons, uses equation (11). Note that while Min-Hash uses the minimum non-zero dimension of a vector as a token, this algorithm uses the bit returned by the $sgn(x)$ function instead. For binary vectors, this algorithm is significantly slower than Min-Hash, even though Min-Hash is intended to be used for Jaccard similarity. This is because for a signature of length R, (11) gives us only 2^R hash buckets, whereas (4) gives us a maximum of d^R hash buckets. Thus, the number of false hash collisions in Min-Hash is much less than that in the other LSH-based algorithm.

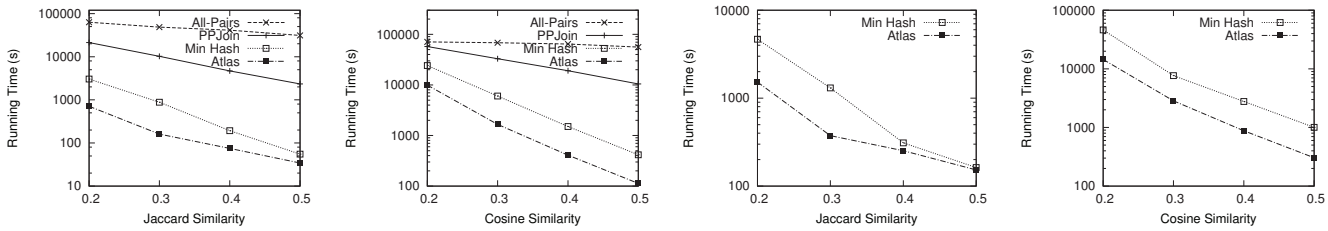
Finally, to illustrate the filtering power of our candidate generation and candidate filtering algorithms, we consider the total number of times that we need to calculate similarities between vector pairs. Figure 5 shows the ratio of the number of candidates generated by the algorithm to the number of vector pairs with similarity above t. The exact algorithms are not included in the comparison since they will have scanned most of the non-zero indices in each vector by the time they finish generating candidates.

Also, note that combining ATLAS's candidate filtering algorithm with Min-Hash *directly* will not significantly improve Min-Hash's performance. Min-Hash is very likely to generate vector pairs with higher similarities multiple times. Since there are more vector pairs with higher similarities in Min-Hash's candidate set, the effectiveness of ATLAS's candidate filtering algorithm is greatly reduced.

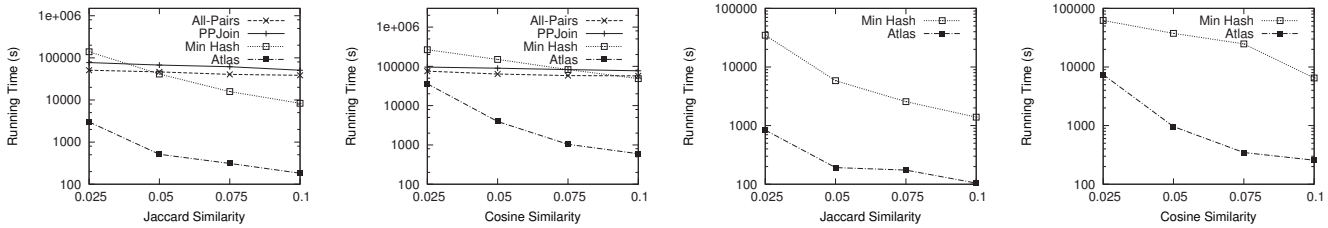
6.3 Vector Pre-Clustering

We evaluated ATLAS's vector pre-clustering algorithm on all four datasets. In all four experiments, we manually set the number of samples to 3,000 and T to 4. We show the the ratio of the number of possible new vector pairs to the number of old vector pairs, $\sum_i |V_i| \cdot (|V_i|-1)/2$ to $|V| \cdot (|V|-1)/2$, the running time, and the number of clusters found in Table 5. We show the recall rates in Table 4.

We find that ATLAS's vector pre-clustering algorithm gives



(a) ClueWeb100K, single core (b) ClueWeb100K, single core (c) ClueWeb, 50 machines (d) ClueWeb, 50 machines



(e) London100K, single core (f) London100K, single core (g) London, 50 machines (h) London, 50 machines

Figure 3: Running Time, Low Similarity Thresholds

ClueWeb100K	0.2	0.3	0.4	0.5
Jaccard	99.61%	99.73%	99.73%	99.85%
Cosine	73.78%	98.97%	99.87%	99.87%
London100K	**0.025**	**0.050**	**0.075**	**0.100**
Jaccard	99.19%	99.79%	99.89%	99.91%
Cosine	93.94%	99.26%	99.64%	99.67%
arXiv	**0.6**	**0.7**	**0.8**	**0.9**
Jaccard	99.81%	99.97%	100.00%	100.00%
Cosine	99.65%	99.77%	99.96%	100.00%
DBLP	**0.7**	**0.8**	**0.9**	
Jaccard	99.10%	99.79%	99.98%	
Cosine	94.68%	98.54%	99.85%	

Table 4: Pre-Clustering: Recall Rates

Dataset		Pairs (%)	Time	Clusters
London100K	Jaccard	45.34%	418.5s	85
	Cosine	67.80%	342.4s	57
ClueWeb100K	Jaccard	16.61%	188.1s	204
	Cosine	28.74%	163.5s	171
arXiv	Jaccard	25.20%	207.2s	81
	Cosine	33.35%	177.2s	63
DBLP	Jaccard	7.95%	1643.7s	246
	Cosine	9.03%	1514.2s	225

Table 5: Pre-clustering: other statistics

good results when t is *relatively* high. Intuitively, when t becomes sufficiently low, vector pairs with similarity above t are distributed more randomly and it becomes harder to capture these vector pairs with clusters. This can be seen from the recall rates for ClueWeb (Cosine, 0.2), London (Cosine, 0.025), and DBLP (Cosine, 0.7). The effectiveness of the algorithm increases as the size of the dataset increases.

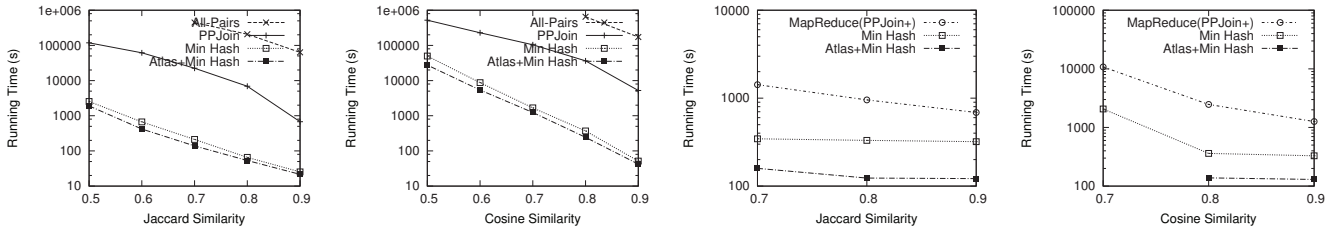
We also integrated ATLAS's vector pre-clustering algorithm with Min-Hash and evaluated it on the arXiv dataset. We used similarity thresholds between 0.5 and 0.9. The results are shown in Figure 6 (a) and 6 (b). We excluded the time needed for vector pre-clustering (around 3 minutes) from the running times as we could use a single clustering result for all five similarity thresholds. Min-Hash offers a significant speed-up (between 10.3x and 132.5x) relative to both PPJoin and All-Pairs. After combining Min-Hash with vector pre-clustering, we gain a further speed-up between 0.18x and 0.81x relative to Min-Hash.

6.4 Scalability

We evaluated the running time of the parallelized algorithms on three datasets: London, ClueWeb and DBLP. The algorithms that we tested include ATLAS, Min-Hash, and Map-Reduce based PPJoin [33]. For ATLAS, we used its candidate generation and candidate filtering algorithms on London and ClueWeb, and its vector pre-clustering algorithm with Min-Hash on DBLP. We did not evaluate All-Pairs and PPJoin as they cannot easily take advantage of parallelism. Instead, we evaluated the Map-Reduce based approach described in [33]. Our parallel implementations of Min-Hash and ATLAS are based on the block-based processing approach that we outlined in Section 5.2.

The Map-Reduce based solution suffers from the memory bottleneck issue [33]. It uses the prefix-filtering idea in [8, 35] to build a *virtual* inverted index. Then, it routes the vectors in each inverted list to a worker for further processing. As we have previously discussed, prefix-filtering is ineffective when the size of the index becomes large. When we allow each mapper/reducer task to use 8 Gbytes of memory, it can only process around 50,000 London records or 400,000 ClueWeb records in one pass. We have to divide the dataset into k blocks, then perform a similarity search on all $k(k-1)/2$ pairs of blocks. Even so, within a 48 hours time limit, this algorithm did not terminate even on ClueWeb ($t = 0.5$) or on London ($t = 0.1$). In contrast, ATLAS finished processing these datasets in 3 minutes for these similarity thresholds.

The results are shown in Figure 3 (c), (d), (g), and (h), and in Figure 6 (c) and (d). Note that the parallelization overhead makes the difference between the speeds of different algorithms appear smaller for higher thresholds. We do



(a) arXiv, single core (b) arXiv, single core (c) DBLP, 10 machines (d) DBLP, 10 machines

Figure 6: Running Time, High Similarity Thresholds.

not include the running times of ATLAS and Min-Hash on the DBLP data with Cosine similarity and a threshold of 0.7, because we used a single clustering result for all experiments which discarded more than 2.5% of the true positives in this case (see Table 4).

7. DISCUSSION

Previous similarity search algorithms either do not address or unsatisfactorily solve the following four problems:

1. Low Similarity Thresholds. We have previously shown in Section 1 that many real-world applications need to use similarity search with low similarity thresholds (0.025 to 0.2 in our examples) and why this makes sense mathematically. Intuitively, we can also see that whenever one is interested in the fine-grained similarities and differences between complex objects (such as in 3D reconstruction [5] and in plagiarism detection [30]), it is hard to find a representation (and a corresponding similarity function) such that the similarity between every interesting pair of objects is high.

2. High Dimensional Descriptors. For feature-rich objects, such as images and videos, the average length of vectors, L, can be very large. The L in our London dataset, which we described in Section 6, is 5,909. Finding an efficient candidate filtering algorithm for these long vectors can make a similarity search algorithm much faster.

3. Incremental Queries. Many real-world applications need to support incremental queries. Examples of such applications include similar music and video retrieval [19, 32], 3D reconstruction [5], and plagiarism detection [30]. In these applications, the user may add a new object to the application while it is running. The application then needs to apply a similarity search algorithm in order to quickly determine all objects that the new object is similar to.

One possible way to deal with newly added objects is to run an all-pairs similarity search algorithm periodically. However, as we have shown in the experiments, recalculating the similarities between all pairs of objects can be very expensive even with only 1 million objects. Similarity search algorithms that support incremental queries can greatly reduce the computational cost.

When new vectors are added, All-Pairs and PPJoin need to resort all vectors and reorder the dimensions, then restart the algorithm from scratch, and thus are not incremental. LSH-based algorithms can support incremental queries, but may need to store a large number of hash tables in memory for low similarity thresholds. Our candidate generation algorithm not only supports incremental queries, but also generally uses less memory than LSH-based methods.

4. Scalability. We have previously discussed the scalability issues in Section 5.2. It is unclear whether exact algorithms can be executed efficiently on a large cluster. As we have shown in Section 6.4, Vernica et al.'s Map-Reduce based solution is ineffective for low similarity thresholds due to its use of prefix-filtering [33]. While LSH-based approximate algorithms can be parallelized using the scheme in Section 5.2, their hash indexes may use a huge amount of memory for low similarity thresholds.

8. RELATED WORK

Exact Algorithms. Many exact algorithms reduce the search space by maintaining an inverted index dynamically. They also use various filtering techniques, such as prefix filtering, positional filtering, suffix filtering, and size filtering [29, 8, 35]. Among those, suffix filtering [35] provides an alternative solution to our candidate filtering problem, but it is effective only when the average length of the vectors is small. These algorithms are highly efficient on smaller datasets. Their drawbacks are mainly (a) the need to preprocess the data, (b) poor performance for low similarity thresholds, and (c) poor scalability as the average length of the vectors increases. Another class of algorithms generates *signatures* for each vector using the pigeonhole principle [7].

Approximate Algorithms. Locality sensitive hashing (LSH) based algorithms hash vectors such that the more similar the two vectors are, the more likely they are to be hashed into the same buckets [25]. In principle, LSH works for a similarity function if and only if the function admits a locality sensitive hash function family. Random projections can be used for Cosine similarity [21, 12], and min-wise independent permutations [11] for Jaccard similarity. In practice, any hash function family that approximately preserves the property of a family of LSH functions can be used to achieve good retrieval performance. Previous work has also used the minimum K values under a hash function for approximate query processing [16, 9], but both their motivation and their usage are quite different from ours.

Dimensionality Reduction. Another approach to process high dimensional vectors is the usage of dimensionality reduction techniques. Popular approaches include principal component analysis (PCA), latent semantic indexing (LSI), singular value decomposition (SVD), and applications of the Johnson-Lindenstrauss Lemma [18]. However, the effects of dimensionality reduction techniques on datasets are highly application-dependent [6].

Partitioning and Clustering. Previous work proposed various partitioning and clustering techniques, but they usually degenerate to linear scans in high dimensions [34]. Clustering methods have also been applied to variants of the

similarity search problem, such as the nearest neighbor problem [13].

9. CONCLUSIONS

We have presented ATLAS, a new probabilistic algorithm for high dimensional similarity search with low similarity thresholds. Our analysis and experimental evaluation suggest that ATLAS is an efficient solution for a wide class of applications.

Future Work. Further improvements to ATLAS's candidate generation procedure are possible. First, we might occasionally experience a low recall rate caused by a bad random seed. A possible remedy is to resort to a *union-intersection-union* procedure like LSH. Second, if the variance in the lengths of the vectors is large, we may need to use a very large K to guarantee a high recall rate. One possible solution is to first run the algorithm to find vector pairs whose differences in lengths are small, then re-run the algorithm with different parameters to find vector pairs whose differences in lengths are large. Third, we still need a large amount of main memory to store the inverted indexes for very low thresholds (in our experiments for the Cosine similarity and a similarity threshold below 0.05).

Acknowledgments. We thank the anonymous reviewers for their valuable comments. This research has been supported by the NSF under Grants IIS-0627680 and IIS-1012593, by the New York State Foundation for Science, Technology, and Innovation under Agreement C050061, and by a Humboldt Research Award from the Alexander von Humboldt Foundation. Any opinions, findings, conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of the sponsors.

10. REFERENCES

[1] http://www.flickr.com/
[2] http://boston.lti.cs.cmu.edu/Data/clueweb09/
[3] http://arxiv.org/
[4] http://dblp.uni-trier.de/xml/dblp.xml.gz
[5] S. Agarwal, N. Snavely, I. Simon, S. M. Seitz, and R. Szeliski. Building Rome in a day. In *ICCV*, 2009.
[6] C. C. Aggarwal. On the effects of dimensionality reduction on high dimensional similarity search. In *PODS*, 2001.
[7] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, 2006.
[8] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, 2007.
[9] K. Beyer, R. Gemulla, P. J. Haas, B. Reinwald, and Y. Sismanis. Distinct-value synopses for multiset operations. *Commun. ACM*, 52(10), 2009.
[10] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *ICDT*, 1999.
[11] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. In *STOC*, 1998.
[12] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.
[13] F. Chierichetti, A. Panconesi, P. Raghavan, M. Sozio, A. Tiberi, and E. Upfal. Finding near neighbors through cluster pruning. In *PODS*, 2007.
[14] O. Chum and J. Matas. Large-scale discovery of spatially related images. *TPAMI*, 32(2), 2010.
[15] V. Chvátal. The tail of the hypergeometric distribution. *Discrete Mathematics*, 25(3), 1979.
[16] C. Cohen and H. Kaplan. Summarizing data using bottom-k sketches. In *PODS*, 2007.
[17] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW*, 2007.
[18] S. Dasgupta and A. Gupta. An elementary proof of the Johnson-Lindenstrauss lemma. Technical Report TR-99-006, ICSI, 1999.
[19] S. L. Feng, R. Manmatha, and V. Lavrenko. Multiple bernoulli relevance models for image and video annotation. In *CVPR*, 2004.
[20] B. J. Frey and D. Dueck. Clustering by passing messages between data points. *Science*, 315(5814), 2007.
[21] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6), 1995.
[22] S. Guha, R. Rastogi, and K. Shim. CURE: A clustering algorithm for large databases. Technical report, Bell Laboratories, 1997.
[23] T. H. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web. In *WebDB*, 2000.
[24] M. R. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, 2006.
[25] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*, 1998.
[26] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 60(2), 2004.
[27] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. In *CVPR*, 2006.
[28] M. Sahami and T. D. Heilman. A web-based kernel function for measuring the similarity of short text snippets. In *WWW*, 2006.
[29] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, 2004.
[30] D. Sorokina, J. Gehrke, S. Warner, and P. Ginsparg. Plagiarism detection in arXiv. In *ICDM*, 2006.
[31] E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating similarity measures: a large-scale study in the Orkut social network. In *KDD*, 2005.
[32] D. Turnbull, L. Barrington, D. Torres, and G. Lanckriet. Towards musical query-by-semantic-description using the CAL500 data set. In *SIGIR*, 2007.
[33] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD*, 2010.
[34] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, 1998.
[35] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, 2008.