Fast Top-K Similarity Queries Via Matrix Compression

Yucheng Low*
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
ylow@cs.cmu.edu

Alice X. Zheng
Microsoft Research, Redmond
1 Microsoft Way
Redmond, WA 98052
alicez@microsoft.com

ABSTRACT

In this paper, we propose a novel method to efficiently compute the top-K most similar items given a query item, where similarity is defined by the set of items that have the highest vector inner products with the query. The task is related to the classical k-Nearest-Neighbor problem, and is widely applicable in a number of domains such as information retrieval, online advertising and collaborative filtering. Our method assumes an in-memory representation of the dataset and is designed to scale to query lengths of 100s to 100,000s of terms. Our algorithm uses a generalized Hölder's inequality to upper bound the inner product with the norms of the constituent vectors. We also propose a novel compression scheme that computes bounds for groups of candidate items, thereby speeding up computation and minimizing memory requirements per query. We conduct extensive experiments on the publicly available Wikipedia dataset, and demonstrate that, with a memory overhead of 21%, our method can provide 1-3 orders of magnitude improvement in query run-time compared to naive methods and state of the art competing methods. Our median top-10 word query time is 25 μs on 7.5 million words and 2.3 million documents.

1. INTRODUCTION

The task of computing the K most similar objects given a query object where similarity is described as distances in a metric space is well known as the k-nearest-neighbor (k-NN) procedure [8]. This procedure is applicable to a wide variety of regression and classification tasks. However, a naive implementation suffers from high computation demands, requiring N distance evaluations for each test data point. To accelerate this procedure, various space partitioning methods such as KD-Trees [6] and M-Trees [12] (among many others) have been proposed, which provide fast exact K-nearest neighbor retrieval [7, 24].

However, when similarity between objects are described by inner products, fast exact top-K retrieval is a much less understood task. Such tasks are common in collaborative filtering (finding similar movies or similar users), similarity queries (search for similar images given a query image) and on-line advertising (display ads that are most textually similar with the current page). Effective solutions here can also generalize to several tasks where kernels are used to define similarity between objects. For instance: graph kernels can be used for similarity search in a molecule database or a gene regulatory network database [29, 28]. Sub-string kernels can be used for document analysis [2] or biological sequence analysis. In these cases, the hash kernel method described in [26] can be used to construct an explicit representation of the feature space thus mapping the task into the top-K inner product regime.

*This work was done while interning at Microsoft Research.

The top-K inner product problem also shares similarities with the top-K query retrieval problem explored heavily by the Information Retrieval (IR) community. Top-K query retrieval techniques include various TAAT (Term-At-A-Time) or DAAT (Document-At-A-Time) procedures [16, 10, 22, 27, 11], which rely on skip-ahead heuristics to quickly iterate through the index and inverted index of the document-word matrix, maintaining an upper bound on each candidate. However, the top-K inner product task we are exploring in this paper differentiates itself from from the query retrieval problem since the query object is itself a datapoint, and thus **can have an unbounded number of terms**. The assumption that the query length is small is not justifiable and the emphasis is thus on scalability to arbitrarily long query lengths without loss of performance.

In this paper we will use text documents as the running example: letting a collection of text documents be represented as a matrix, where each row represents a document and each column a word. The (i, j)-th entry is the count of the number of times the j-th word appears in the i-th document. The similarity between words w_1 and w_2 can then be defined as the inner product of the two column vectors w_1 and w_2:[1]

$$\text{Similarity}(w_1, w_2) = w_1^T w_2.$$

This is also the co-occurrence count of how many times w_1 and w_2 both appear in the same document. Analogously, the similarity between documents d_1 and d_2 is the inner product between the row vectors:

$$\text{Similarity}(d_1, d_2) = d_1 d_2^T.$$

Letting D denote the document-word matrix, the task of querying the K most similar words to a query word v is equivalent to computing the K largest elements of the row vector $v^T D$; and the most similar documents to a query document q is gleaned from the column vector $D^T q$. Note that in both cases the computation are essentially identical, but are performed on either the document-word matrix or its transpose.

In this paper, we present an efficient algorithm that computes the *exact top-K* largest inner products given a query word [2]. Such a system for instance, could be used to find "related words" in a user search. The rationale for using words as query objects instead of documents, is also to provide a wider range of query lengths, allowing us to benchmark the system on both extremely short and extremely long queries. We focus strictly on the in-memory set-

[1]With a slight abuse of notation, we shall use w to denote both the query word and its corresponding vector representation.

[2]For readers familiar with DAAT / TAAT, note that our presentation in this paper is worded for the transposed problem, i.e., word-word top-K queries. However, we do report on experiments for both word and document top-K similarity queries.

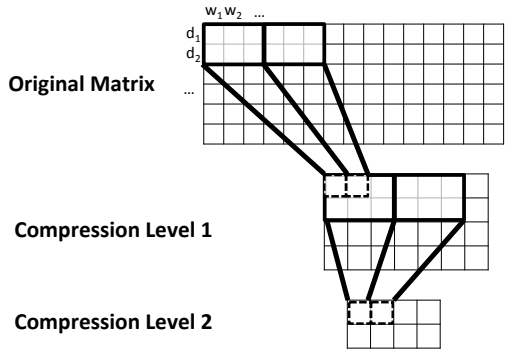


Figure 1: A document collection is represented as a sparse matrix where entry (i, j) in the matrix counts the number of times document i contains word j. More generally, the matrix elements could contain arbitrary values such as *tf-idf* scores. The document-word matrix is then compressed into a smaller matrix. Each element in the smaller matrix represents a block in the original matrix.

ting, exploring the performance characteristics of different matrix representation formats.

Our method augments the matrix with an additional data structure containing compressed summaries of the document-word matrix. We demonstrate that our technique scales well to extremely long query lengths, and provides *1-3 orders of magnitude* performance gains over competing methods for all query lengths, far exceeding the gains reported by memory-based variants of DAAT / TAAT algorithms even for short queries. [16] Our method incurs a memory overhead that depends on the statistics of the underlying dataset and a tunable algorithm parameter, allowing the user to balance between query speed and memory utilization. For the query time results previewed above, we observe a 21% memory overhead over a set of 1 million Wikipedia articles. On the full dataset of 2.3M articles and 7.5M words, our median top-10 query time for a randomly selected word query is 25 μs.

In the following sections, we describe our algorithm in detail (Sec. 2), proving its various properties along the way. Implementation details are discussed in Sec. 3, and experimental results in Sec. 4. We conclude with a discussion of relationships to other algorithms (Sec. 5.1), parallelization issues (Sec. 5), and future work (Sec. 6).

2. ALGORITHM

Given a query word v, our goal is to quickly compute a list of its top-K most similar words, i.e., the K words with the largest inner product (co-occurrence count) with v. (Note that the popular cosine similarity measure is a variant of inner product similarity and can thus be computed using the same algorithm.) A naive algorithm for doing so may proceed thus: first compute an upper bound of co-occurrence for every word with every other word in D. Then, at run-time, sort the words by upper bounds, refine the upper bound for the top candidate, re-insert it back into a max-heap, and repeat. The algorithm terminates when the first K elements in the heap are exact co-occurrence counts as opposed to upper bounds. A template for this algorithm is shown in Alg. 1.

Making this algorithm efficient requires three key conditions.

1. Ideally, the upper bound should be tight, i.e., it should be as small as possible while remaining an upper bound.
2. The upper bound should be significantly faster to compute than the actual inner product.

Algorithm 1: Top-K Algorithm Template

```
Input: v : Query Word
Input: ubnd(v, w) : A function which upper bounds v^T w

H : max heap of (word, value) pairs
foreach w ∈ W do                    // Upper Bound each Word
    Insert (w, ubnd(v, w)) into H
ReturnWords = {}
while |H| > 0 do
    Pop (w, val) from H
    if val was computed using ubnd() then
        // This is an upper bound.  Get true
           value by computing the inner product
        Insert (w, v^T w) into H
    else
        // This is true value.  Since it is
           greater than all other upper bounds,
           it is in the top K
        Insert w into ReturnWords
        if |ReturnWords| = K then return ReturnWords
return ReturnWords
```

3. The maintenance of a large W sized heap in Alg. 1 is inefficient. It is necessary to reason about collections of words at the same time.

There is also a fourth, more subtle condition which is necessary for consistent performance over a broad scale of datasets and query lengths. While it is natural for the algorithm to balance between the quality of the upper bound against computational cost, it is also important that the algorithmic complexity scales *at most linearly* with the length of query, and *at most linearly* with the size of the dataset. For instance, the computation of word-word co-occurrence counts require the algorithm to operate efficiently even on queries with over 100,000 terms (Table 3). The mWand algorithm [10, 16] in particular requires a costly sorting routine which results in a sharp increase in runtime as query length increases (Fig. 5(a)).

Our solution, which satisfies all key conditions of efficiency, involves constructing a "compressed" auxiliary matrix containing upper bounds of chunks of words and/or documents of the original document-word matrix. The upper bounds are established via an extension of Hölder's inequality to the case of generalized matrix norms. We call the algorithm HComp, for Hölder Compression. To facilitate the explanation, we begin with an overview of Hölder's inequality and how it can be used to provide an upper bounding heuristic.

2.1 Review of Hölder's inequality

Hölder's inequality upper bounds the inner product of two vectors by the product of their norms. Specifically, for any vectors a and b and scalars p and q such that $1 \leq p, q \leq \infty$ and $1/p + 1/q = 1$, Hölder's inequality states that

$$a^T b \leq ||a||_p ||b||_q, \tag{2.1}$$

where $||x||_p = (\sum_i |x_i|^p)^{1/p}$ is known as the p-norm of x. In this paper, we make use of the 1-norm, 2-norm (e.g. the Euclidean norm), and the infinity norm.

$$\begin{aligned} ||x||_1 &= \sum_i |x_i| \\ ||x||_2 &= \sqrt{\sum_i x_i^2} \\ ||x||_\infty &= \max_i |x_i| \end{aligned}$$

Note that Hölder's inequality yields not just one but a family of upper bounds, i.e., it is valid for any combinations of p and q satisfying $p \geq 1$ and $1/p + 1/q = 1$. Common examples of such combinations include $(p = q = 2)$, $(p = 1, q = \infty)$, and $(p = \infty, q = 1)$. That is to say,

$$|a^T b| \leq ||a||_2||b||_2 \quad (2.2)$$

$$|a^T b| \leq ||a||_1||b||_\infty \quad (2.3)$$

$$|a^T b| \leq ||a||_\infty||b||_1 \quad (2.4)$$

Eq. (2.2) is also known as the Cauchy-Schwarz inequality. A natural question to ask is whether any of these bounds are provably tighter than the rest. The answer is yes for binary vectors, an important class of word vectors under our consideration.

LEMMA 2.1. *Let a and b be arbitrary binary vectors, i.e., a_i is either 0 or 1, for all i, and the same for b_i. Then the minimum Hölder upper bound for $a^T b$ occurs at either $p = 1, q = \infty$ or $p = \infty, q = 1$. In other words,*

$$a^T b \leq ||a||_1||b||_\infty \leq ||a||_p||b||_q, \quad \textit{for all } p \neq 1, q \neq \infty,$$

or

$$a^T b \leq ||a||_\infty||b||_1 \leq ||a||_p||b||_q, \quad \textit{for all } p \neq \infty, q \neq 1.$$

Furthermore, let n_a and n_b respectively denote the number of non-zero elements in a and b. If $n_a > 0$ and $n_b > 0$, i.e., a and b are not all-zero vectors, then the minimum Hölder bound is $\min(n_a, n_b)$.

PROOF. Hölder's inequality dictates that $1/p + 1/q = 1, p \geq 1$. Fixing a value for p and solving the constraint for q yields $q = p/(p-1)$. We plug this value back into the upper bound and solve for

$$\hat{p} = \arg\min_p ||a||_p||b||_{p/(p-1)}.$$

Let us denote $f(p) = ||a||_p||b||_{p/(p-1)}$. Suppose a has n_a non-zero entries and b has n_b non-zero entries. Then

$$\begin{aligned} f(p) &= \left(\sum_i a_i^p\right)^{1/p} \left(\sum_i b_i^{p/(p-1)}\right)^{(p-1)/p} \\ &= (n_a)^{1/p}(n_b)^{(p-1)/p}. \end{aligned}$$

To make the minimum more obvious, we take the logarithm of the function—a monotonic transformation that does not change the location of the minimum.

$$\log(f(p)) = \frac{1}{p}\log(n_a) + \frac{p-1}{p}\log(n_b).$$

Note that this is simply a convex sum of $\log(n_a)$ and $\log(n_b)$. In fact, letting p sweep from 1 to ∞ will interpolate smoothly between $\log(n_a)$ and $\log(n_b)$. The minimum of the sum occurs when all weight is placed on the smaller of the two extremes

$$\min\log(f(p)) = \min(\log(n_a), \log(n_b)),$$

which is achieved when $p = 1$ or $p = \infty$. □

Thus, the best upper bound for the inner product of binary vectors is achieved by either $p = 1, q = \infty$ or $q = 1, p = \infty$. This is confirmed by our empirical observations as well. This bound is also tight. Consider the case where $b_i = 1$ for all i, i.e., the word appears in every document. Then the co-occurrence count is equal to n_a. In this case, setting $p = 1$ yields an exact upper bound:

$$n_a = a^T b \leq ||a||_1||b||_\infty = n_a \cdot 1 = n_a.$$

Thus, the $1 - \infty$ mixed-norm bound is the optimal Hölder bound, and is the best one can hope for for certain binary word vectors.

2.2 Upper Bounding with Hölder's Inequality

Hölder's inequality provides a conceptually simple method of providing upper bounds to every word-word inner product by simply storing a "norm" for each word. For instance, if we choose to use the $(p = 1, q = \infty)$ bound, we can simply pre-compute and store the value of $||a||_1$ and $||a||_\infty$ for each word $a \in W$.[3]

However, it is easy to see that this technique will not be very effective in general. Given a choice of p and q that satisfies Hölder's inequality, if we define an upper bounding function for Alg. 1 such that for a query word v,

$$\textbf{ubnd}(v, w) = ||v||_p||w||_q,$$

then, **ubnd(v,w)** is proportional to $||w||_q$ for any fixed query word v. This means that the search order in Alg. 1 will simply be in order of decreasing $||w||_q$. In other words, if we choose $(p = \infty, q = 1)$ as above, the search order will simply be in decreasing word frequency, regardless of the query. While this is a reasonable top-K search heuristic (under the intuition that frequent words are also likely to co-occur frequently with any given query word), it is not very efficient in practice. A method which can integrate more "contextual" information about the query word v is desirable. Furthermore, this simple upper bounding method does not provide a way to satisfy condition 3–reasoning about collections of words simultaneously.

2.3 Faster Bounds Via Matrix Compression

The solution we propose in this paper is to compress the document-word matrix into a smaller matrix, each element of which is an upper bound for an entire block of the original matrix (Fig. 1). Specifically, the element in the smaller matrix is a generalized matrix norm–a provable upper bound of any column or row norm–of the original matrix block.

We start by defining two generalized matrix norms. Let $A \in \mathbb{R}_+^{m,n}$ be a matrix of m rows and n columns whose elements are non-negative real numbers. Let a_{ij} denote the (i, j)-th element of A (i.e., the element at the i-th row and the j-th column). We define the u-v mixed norm of the matrix A as a function $L^c_{\alpha,\beta}(A)$, where $\alpha \geq 1, \beta \geq 1$,[4] and

$$L^c_{\alpha,\beta}(A) = \left(\sum_j (\sum_i a_{ij}^\alpha)^{\beta/\alpha}\right)^{1/\beta}. \quad (2.5)$$

Essentially, $L^c_{\alpha,\beta}(A)$ computes the α-norm of each column, then computes the β-norm of the result. We also define the transposed mixed-norm $L^r_{\beta,\alpha}(A)$ that first computes the row norms, then the norm of the resulting column:

$$L^r_{\beta,\alpha}(A) = \left(\sum_i (\sum_j a_{ij}^\beta)^{\alpha/\beta}\right)^{1/\alpha}. \quad (2.6)$$

Both L^r and L^c are upper bounds of the α-norm of every column of A. In other words, the ordering in which we "collapse" the matrix does not affect the fact that the resulting scalar is an upper bound of the norm of any column. Furthermore, we show that for any α, setting $\beta = \infty$ gives us the tightest upper bound.

[3] Another advantage of the p-norms is that they can be easily updated online. This is useful when documents are delivered in a stream. For example, $||a||_1$ is simply the total number of occurrences of the word in all documents and can be updated via the summation operator. $||a||_\infty$ is simply the maximum number of times the word is seen in any given document and can be updated via the max operator.

[4] Even though these definitions require $\alpha, \beta < \infty$, analogous forms for the ∞-norm (max-norm) can be defined.

LEMMA 2.2. *Let a_j denote the j-th column of A. We have, for any j and any $\alpha \geq 1$,*

$$||a_j||_\alpha \leq L^c_{\alpha,\beta}(A) \tag{2.7}$$

and

$$||a_j||_\alpha \leq L^r_{\beta,\alpha}(A). \tag{2.8}$$

Furthermore, for a fixed α, $L^c_{\alpha,\infty}$ provides the tightest generalized matrix norm upper bound. That is to say, for any $\beta \neq \infty$, we have

$$L^c_{\alpha,\infty}(A) \leq L^c_{\alpha,\beta}(A), \tag{2.9}$$
$$L^r_{\infty,\alpha}(A) \leq L^r_{\beta,\alpha}(A), \tag{2.10}$$
$$L^c_{\alpha,\infty}(A) \leq L^r_{\infty,\alpha}(A). \tag{2.11}$$

PROOF. From the definition of $L^c_{\alpha,\beta}$ in Eq. (2.5), and assuming $\alpha, \beta < \infty$, we can easily show that

$$\begin{aligned} L^c_{\alpha,\beta}(A) &= (||a_1||^\beta_\alpha + ||a_2||^\beta_\alpha + \ldots + ||a_n||^\beta_\alpha)^{1/\beta} \\ &\geq (||a_j||_\alpha)^{\beta/\beta} \\ &= ||a_j||_\alpha \end{aligned}$$

for any j, because the norm of a vector of non-negative elements is always greater than each individual element. For $L^r_{\beta,\alpha}$, we have, for any fixed j,

$$\begin{aligned} L^r_{\beta,\alpha}(A) &= (\sum_i (\sum_\ell |a_{i\ell}|^\beta)^{\alpha/\beta})^{1/\alpha} \\ &\geq (\sum_i (|a_{ij}|^\beta)^{\alpha/\beta})^{1/\alpha} \\ &= (\sum_i |a_{ij}|^\alpha)^{1/\alpha} \\ &= ||a_j||_\alpha \end{aligned}$$

where again we've used the fact that the sum of a number of non-negative elements is greater than any individual element. Proofs for the case where $\alpha = \infty$ or $\beta = \infty$ are similar. This proves Eqs. (2.7) and (2.8).

Of the rest of the claims, Eqs. (2.9) and (2.10) are simple consequences of the definitions of the norms and the fact that the ∞-norm is the smallest of all p-norms of any vector. To prove Eq. (2.11), let t be the column vector formed by taking the maximum of each row of A, then

$$\begin{aligned} L^c_{\infty,\alpha}(A) &= \max_i((\sum_j |a_{ji}|^\alpha)^{1/\alpha}) \\ &\leq \max_i((\sum_j |t_j|^\alpha)^{1/\alpha}) \\ &= \max_i(||t||_\alpha) = ||t||_\alpha = L^r_{\infty,\alpha}(A). \end{aligned}$$

Intuitively, $L^r_{\infty,\alpha}$ takes the norm of a column vector produced by the maximum absolute value of each row, and $L^c_{\alpha,\infty}$ takes the maximum of each column norm. Since the entries in each column of the matrix are necessarily smaller than the column vector computed by $L^r_{\infty,\alpha}$, the final norm is also necessarily smaller. □

Combining Hölder's inequality (Eq. (2.1)) and Lemma 2.2, we obtain an upper bound on a collection of inner products.

THEOREM 2.3. *Given a query vector v and a matrix A. Then for any column vector a_j in A and for any p, q satisfying the conditions of Hölder's inequality:*

$$v^T a_j \leq ||v||_p L^c_{q,\infty}(A), \quad \textit{for all } j$$

Theorem 2.3 allows us to compress the entire matrix A to yield an upper bound. Clearly, the quality of the bound degrades when the matrix is bigger, e.g., when we "compress" more columns (words) and/or more rows (documents) together. Therefore there is a trade-off between bound quality and computation efficiency. We examine this trade-off empirically in Sec. 4.1.1. Furthermore, we observe that the procedure permits multiple levels of compression since the compressed matrix can be further compressed into an even smaller matrix. The double-compressed matrix still retains the property that it is an upper bound on blocks in the original matrix. The following corollary summarizes the results of this section.

COROLLARY 2.4. *Given query vector v and document-word matrix D, let $\tilde{v}$ denote the p-compression of the vector v, and $\tilde{D}$ denote the q-compression of the matrix D. Let d_j denote the j-th column of D, and $\tilde{D}_{(j)}$ the corresponding compressed blocks. Then*

$$v^T d_j \leq \tilde{v}\tilde{D}_{(j)}.$$

Moreover, $\tilde{v}$ and $\tilde{D}$ can be further compressed to yield looser but more computationally efficient upper bounds.

2.4 HComp Algorithm

We now provide a complete definition of the complete HComp algorithm which operates on a hierarchy of compressed matrices. Firstly, a pair of p, q that satisfies Hölder's inequality are chosen. Next, a row compression factor r and a column compression factor s are chosen. The optimal choice of r and s depends heavily on the both the dataset and the properties of the underlying matrix representation. Then, given an input matrix D, the `CompressMatrix` function in Alg. 2 is used to generate the (possibly hierarchical) compression. For notational convenience, the algorithm insets in this paper use dense matrix representations with direct block indexing. However, the algorithms are easily modified to manage sparse matrix representations. The sparse matrix representation is much more efficient in memory and is what we use in our implementation.

At query time, the query word/vector v is similarly hierarchically compressed using the `CompressVector` function in Alg. 3. Then the top-K algorithm in Alg. 4 is called. The algorithm in Alg. 4 begins by using the coarsest compression of the original matrix to provide upper bounds on *ranges of words* which are then stored in a max-heap. Elements are then popped from the heap, and if the element is a range, the range is refined by expanding it to the next finer compression level, effectively splitting the range into a series of smaller ranges. If the element is a single word, it must be larger than all other upper bounds and therefore belong in the top K set.

2.5 Key Properties

The following properties of the HComp algorithm may be observed.

Flexibility in Matrix Representation: The only additional requirement the HComp algorithm has on the matrix representation is the ability to compute $v^T D^{(\mathbf{c})}$ where $D^{(\mathbf{c})}$ is a contiguous range of columns in D. This is easily provided by most in-memory matrix representations.

Incremental Construction: While the compressed matrix can be easily constructed off-line, an on-line incremental construction is possible for some choices of norms. For instance, when $q = \infty$, the compression only requires taking maximum of blocks, and thus can be easily constructed incrementally.

Query Memory Utilization: The amount of additional memory required at query time is proportionate to the size of the max-heap produced in Alg. 4. The number of heap elements required is dependent on the properties of the dataset, as well as the quality of

Algorithm 2: CompressMatrix(D, lvls, q, r, s): Hierarchical Compression of matrix A

Input: D : matrix to compress
Input: lvls : Number of compression levels. If lvls = 0, no compression is performed
Input: q : Choice of column norm
Input: r : height of the compression block.
Input: s : width of the compression block.
if *lvls = 0* **then return** *[D]*
Make empty matrix D'. This will be the compression of D
// D' has height $\lceil$height of D/$r\rceil$ and width $\lceil$width of D/$s\rceil$
for $i = 1$ *to height of* D' **do**
  for $j = 1$ *to width of* D' **do**
    Let C be the submatrix in D associated with $D'_{i,j}$
    $D'_{\text{i,j}} = L^c_{q,\infty}(C)$
// Recursively Compress D'. Final output will be a list of successive compressions of D.
return *[D, Compress(D', lvls -1, q, r, s)]*

Algorithm 3: CompressVector(w, lvls, q, r): Hierarchical Compression of query column vector w

Input: v : vector to compress
Input: lvls : Number of compression levels. If lvls = 0, no compression is performed
Input: p : Choice of column norm
Input: r : height of the compression block.
return *CompressMatrix(v, lvls, p, r, 1)*

Algorithm 4: HComp(): HComp Top-K Algorithm

Input: lvls : Total number of levels of compression
Input: $D_0 \cdots D_{\text{lvls}}$ where D_0 is the full document-word matrix and the rest are successively higher levels of compression using a $L^c_{q,\infty}$ mixed-norm
Input: $v_0 \cdots v_{\text{lvls}}$: where v_0 is the full query word vector, and the rest are successively higher levels of compression using a p-norm where p, q satisfies Hölder's condition
ReturnWords = {}
// [colidx, level] identifies a specific column at a compression level. When level is 0, the column represents a single word
H : max heap of ($[colidx, level]$, value) pairs
// Construct initial bound using highest level
FirstBound = $w^T_{\text{lvls}} D_{\text{lvls}}$
for $i = 1$ *to* $|$*FirstBound*$|$ **do**
  Insert ($[i,$ lvls$]$, FirstBound$_i$) into H
while $|H| > 0$ **do**
  Pop ($[$idx$, l]$, val) from H
  if $l > 0$ **then**
    $l' = l - 1$ // refine by one level
    Let **c** be the range of columns in level $l-1$ associated with column idx in level l
    for i *in* **c** **do**
      Insert $([i, l'], v^T_{l'} D_{l'}{}^{(i)})$ into H
  else if $l = 0$ **then**
    Insert idx into ReturnWords
    if $|$*ReturnWords*$| = K$ **then return** *ReturnWords*
return *ReturnWords*

the upper bounds produced. Even though in the worst case the size of the heap can be as large as the number of words in the dictionary, in practice we find that the heap size is quite small requiring only 1,000s of entries for a dictionary of 4.6M words. We examine empirical query memory utilization in Sec. 4.2.

Quality of bounds: Intuitively, the matrix compression strategy produces the tightest upper bounds when a dense block of elements are mapped into a single compressed element. In other words, we obtain good bounds when words which co-occur frequently together, or documents with many similar words, are "chunked" together in a single block. The optimal arrangement of blocks is thus to group together frequently co-occurring words, and to group together similar documents. However, optimizing the block arrangement requires co-occurrence counts, which are exactly what we were trying to solve in the first place!

However, we may observe that when the matrices are constructed incrementally, i.e., as documents stream in and the word index is ordered by their first appearance, then frequent words are naturally clustered together, as are rare words. Common words will naturally cluster towards the lower word IDs while infrequent words will be assigned higher word IDs. This procedure therefore provides a simple approximation to the block arrangement problem. As a result, the incremental construction procedure permitted by the design of the HComp algorithm provides an additional advantage by improving the quality of the bounds produced. This is the strategy we adopt in our implementation of HComp.

3. IMPLEMENTATION

We implemented the HComp algorithm above under two different matrix representation formats, the **Jagged Column Store** as well as the **Jagged Row and Column Store**. The implementation is written in C++, using standard STL containers. Both documents and words are identified by sequential 32-bit integers. Values in the matrix are also represented as 32-bit integers. `libboost`'s `unordered_map` is used as a hash table when needed.

We make the choice of $(p = 1, q = \infty)$ for the HComp algorithm. This choice is partly motivated by the result in Lemma 2.1 that suggests that the $(1, \infty)$ pair is optimal for binary data. It is also motivated by computational convenience: the matrix compression procedure in Alg. 2 using the $L^c_{q,\infty} = L^c_{\infty,\infty}$ norm simply computes the maximum value of blocks in the document-word matrix.

3.1 In-Memory Matrix Representations

Jagged Column Store: In the Jagged Column Array representation, the document-word matrix is represented as a vector of word vectors, where each word vector is a sorted vector over documents containing the word. A graphical example is shown in Fig. 2(a). This representation has the advantage that it requires very little memory to store. We will use the acronym **CS** to identify algorithms implemented with the Jagged Column Store. These word vectors are known as postings lists in the DAAT / TAAT literature.

Jagged Row And Column Store: In this representation, the document-word matrix is represented in both column format (Fig. 2(a)) and row format (Fig. 2(b)). The row representation essentially acts as an inverted index for the column representation. We will use the acronym **R&CS** to identify algorithms implemented with the Jagged Row and Column Store.

3.2 Data Ingress

We assume that the data is organized so that ingress into memory can be performed one document at a time. Both matrix representations are computationally efficient and permit linear time insertion of documents.

The hierarchical compression of the matrix is performed as a final pre-processing stage and has complexity linear in the number of non-zero matrix elements (**NNZ**), thus requiring only one pass through the data. Each higher level matrix uses the same matrix storage format as the lower levels. Thus when HComp is evaluated

on the CS representation, all higher compression levels are also stored as CS.

3.3 Naive Top-K Algorithm

As a baseline for both matrix representations, we first implement the naive approach to computing top-k word co-occurrence. Given the document-word matrix D and a query vector v, the naive approach simply computes the entire matrix vector product $v^T D$, returning the top-K entries in the resultant vector.

Computing $v^T D$ in the CS representation requires the computation of sparse inner products of v against every column in D. Since both vectors are sorted by the document ID, the algorithm proceeds by iterating through both v and a column in D in parallel, accumulating the inner product sum. This is repeated for each column.

Computing $v^T D$ with the reverse index in the R&CS representation is simpler. Since v contains a list of all documents containing the query word, all inner products may be computed by iterating through the documents in d and accumulating a sum of the corresponding document vector in D, weighted by the entry value in v. Our implementation stores the resulting sum of vectors in a hash table, avoiding the computational cost of constructing a sparse vector. The computation cost is therefore only linear in the total number of words in all documents containing the query word v.

3.4 mWand

We also implement the Wand algorithm described in Broder et al. [10] with the in-memory mWand optimization in Fontoura et al. [16]. Wand is a popular method in the IR community that is the most directly comparable to our method. While it is not designed for long query lengths, it provides a reliable evaluation baseline.

The Wand Iterator uses the inverted index and provides a clever upper bounding strategy which allows it to skip and ignore some of the entries in the inverted index. The Wand Iterator also uses a threshold value which allows it to perform approximate top-K. For the purposes of this evaluation, we set the threshold value to 0 which forces exact top-K evaluation. Since the mWand algorithm requires the inverted index, it can only be implemented using the R&CS store.

3.5 HComp Top-K Algorithm

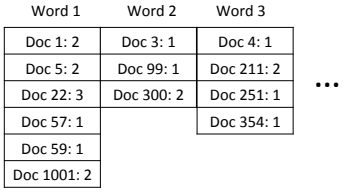
As noted in Sec. 2.5, the HComp algorithm only requires the matrix representation to provide the ability compute inner products against contiguous ranges of columns in D. This is trivially provided by the **Jagged Column** representation reusing the sparse inner product procedure as described in Sec. 3.3.

In the **Jagged Row and Column** representation, we sum over the set of documents in v restricted to the columns in **c**. Since the Jagged Row representation stores each document as a sparse sorted list of columns, restricting the sum to a contiguous range of columns requires an additional binary search to locate the start of the range.

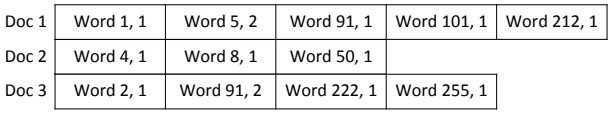
4. EVALUATION

To evaluate HComp, we use a set of 1 million randomly selected Wikipedia articles with common stop words removed. The statistics of our 1M wikipedia dataset are shown in Table 1. For comparison, we also provide the statistics of the large dataset in Fontoura et al. [16] in Table 2. (This dataset is not publicly available for testing.)

Compared to the dataset evaluated in Fontoura et al. [16], which has 454M non-zero elements, the Wikipedia dataset is roughly half the size. However, the 1M Wikipedia dataset has a significantly smaller ratio of unique words to documents (4.6M : 1M) as compared to (69M : 3.5M) in Fontoura et al. As a result, the average



(a) Jagged Column Array



(b) Jagged Row Array

Figure 2: (a) Jagged column representation of the document-word matrix. (b) Jagged row representation of the document word matrix.

Rows (Documents)	1,000,000
Columns (Unique Words)	4,604,909
Non-Zero Elements	217,426,595

	Mean	**Stddev.**	**Median**
Words Per Doc	501.35	820.45	252
Unique Words Per Doc	162.74	209.89	99
Documents Per Word	55.2	1,911.77	1
Unique Documents Per Word	35.35	1,011.02	1

Table 1: Wikipedia Dataset Statistics

number of *Unique Documents Per Word* is significantly larger in the Wikipedia dataset at 35.3 vs 6.53, and the average number of Unique Words per document is also larger at 162.74 vs 130.33.

Our first task involves solving the "transposed" problem as compared to Fontoura et al. Thus *Words Per Document* in one table should be compared against *Documents Per Word* in the other table. The problem statistics are therefore significantly different. In particular, we have a smaller average number of unique terms per candidate (35.35 vs 130.33), but with a significantly larger standard deviation (1011.02 vs 103.86); our task involves a much larger variation in the sizes of the candidates. This difference in dataset statistics explains our unusual performance results on this task.

For our second task in Sec. 4.3, we examine the transposed problem which is more directly comparable to the experimental setup in Fontoura et al., though the performance gap still exists.

4.1 Top-K Word Co-occurrence Experiment

We generate a query set by extracting 10,000 random columns from the matrix. The statistics of the query set are shown in Table 3. In comparison to the "large query" dataset in Fontoura et al., our query set has **10x** the average query length. The extremely large standard deviation in our query set (4,897) as compared to Fontoura et al. (3.32) shows that we have queries with massive number of terms. Indeed, our longest query has 203,248 terms.

Our evaluation task is to return the exact top-10 other words co-occuring with the words in the query set. Essentially, where D is the document-word matrix, and v is a query comprising of a column extracted from D, the task is to return the top 10 entries of $v^T D$. We evaluate the in-memory performance of the HComp algorithm, comparing against naive strategies for both CS and R&CS matrix representations. We also evaluate against the mWand iterator algorithm configured to return exact top-10. We use the g++

Rows (Documents)	3,485,597
Columns (Unique Words)	69,593,249
Non-Zero Elements	454M

	Mean	Stddev.
Unique Words Per Doc	130.33	103.86
Unique Documents Per Word	6.53	1,212.95

Table 2: Fontoura et al. Large Dataset Statistics

Number of Queries	10,000
Mean Query Length	563.02
Stddev. Query Length	4,897
Median Query Length	16
Largest Query Length	203,248

Table 3: Query Statistics. Query Length is the number of non-zero entries in the query vector.

4.1.2 compiler with the `-O3` compiler option running on a Intel(R) Xeon(R) CPU E5620 2.40GHz machine with 16 GB of RAM.

4.1.1 Effect Of Compression Block Shape

Since the effect of varying the compression block size and shape can be dataset dependent, we first evaluate the optimal choice of compression block shape. We do so by computing the median runtime of random queries on a 5% subsample of the documents varying the size and shape of the compression block. For this experiment, we use only one compression level.

Fig. 3(a) and Fig. 3(b) plot the median runtime of varying both compression block size and compression block shape, respectively for CS and R&CS representations. In both figures, a line denotes a single "compression factor," equal to the width $\times$ height of the block in the original matrix. The X-axis denotes the width of the block (number of words compressed); the height of the block (number of documents compressed) is understood to be whatever is required to maintain a constant compression factor.

We observe from Fig. 3(a) that the choice of compression factor does not impact performance under the CS representation. (Note the small scale of the Y-axis.) Neither are there are obvious trends based on the shape of the blocks. However, the R&CS representation is *significantly* faster with smaller compression block sizes and wider block shapes (more words than documents). This behavior is expected since smaller block sizes have the effect of improving the quality of the upper bound. However, performance will degrade when the block size shrinks too far as the cost of computing the upper bounds approaches the cost of computing a full naive query. The advantages of wider block sizes is also easy to understand. As stated in Sec. 3.5, unlike the CS representation, expanding a range in the R&CS representation incurs a cost of an additional binary search. Wider block shapes thus decrease the number of binary searches required by maximizing the number of expanded words in each search. Of course, excessively wide blocks would degrade the quality of the upper bound and result in performance loss.

For the remaining experiments, we pick a compression ratio of 1000:1 and map 1 doc $\times$ 1000 words into a single compressed entry–the fastest parameter setting for both CS and R&CS representations.

4.1.2 Memory Overhead of Compressed Matrices

Next, we evaluate the memory overhead of maintaining the compressed matrices. In all cases we choose to compress only in the dimension of words, i.e., the blocks contain a single document and as many words as is the compression factor. Since there are only 3.5M unique words, only up to 2 levels of compression are meaningful at a compression block size of 1 doc $\times$ 1000 words: a third level of compression will be a single column vector. For the 5000:1 and 10000:1 cases, level 2 compression already results in a single column vector. Table 4 presents comparison results of memory overhead and the median run time of 10K random queries. With a 2-level, 1000:1 compression, the memory overhead is only 21%, while median query time is 28 times faster than the baseline naive algorithm with no compression. More aggressive compression reduces memory overhead, at a small cost to query performance.

Compression Factor	% Overhead	Med Query Time (ms)
1:1	0%	2.46
1×100:1	53%	0.0959
1×500:1	28%	0.0721
1×1000:1	21%	0.0869
1×5000:1	11%	0.163
1×10000:1	8.6%	0.214

Table 4: Comparison of memory overhead and median query run time at different compression factors. All results are obtained with block shapes of a single document and the R&CS representation. The first line represents the native algorithm with no compression.

For both matrix representation strategies, we evaluate the effectiveness of using hierarchical compression. Fig. 4(a) plots the query performance as the number of compression levels are increased when the CS matrix representation is used. Fig. 4(b) provides the same but for the R&CS representation. We observe that, for the CS representation, adding the first level of compression provides significant performance gains, but adding the second level provides almost no performance gain for short queries, and only a small gain for large queries. On the other hand, the R&CS representation demonstrates consistent uniform performance gains across all query lengths as the compression hierarchy is increased.

4.1.3 Combined Results

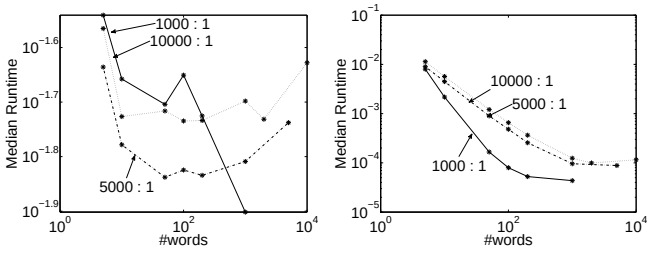
Finally, in Fig. 5(a) we plot the combined performance of all matrix representations for all algorithms: Naive, HComp and mWand. We summarize our observations here:

mWand vs CS: We observe that the mWand algorithm is faster than Naive CS by about 10x for all queries, and is also faster that HComp CS for short queries. But when the query length exceeds 1000, the overhead of the mWand algorithm starts to become evident. For extremely long queries of 10^4 to 10^5 terms, the HComp CS algorithm, can be faster by up to 2 orders of magnitude.

HComp CS vs Naive CS: The HComp CS algorithm provides a small performance gain on small queries (about the same as the mWand algorithm), but as query length increases, the performance gain widens to *2-3 orders of magnitude* for queries with length exceeding 10,000.

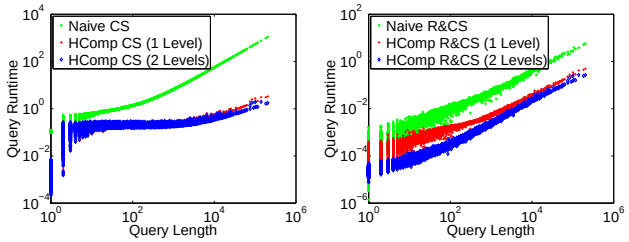
Naive R&CS vs mWand and Naive CS: Here our implementation of the Naive R&CS algorithm is able to significantly outperform the both the mWand algorithm and the Naive CS by a consistent 1 order of magnitude across all query lengths. As far as the authors are aware, this performance gap is unprecedented and has not been observed in information retrieval literature to date. (The performance gap between naive DAAT and naive TAAT is typically not so large.) However, as we see in Sec. 4.3, this can be attributed to the statistics of the transposed problem in this paper–our main task is to find similar *words* and not *documents*.

HComp R&CS vs Naive R&CS: HComp provides a consistent 1 order of magnitude performance gain over all query lengths above the Naive method using the R&CS representation. This gain is consistent on the transposed problem (see Sec. 4.3).



(a) CS Varying Shape (b) R&CS Varying Shape

Figure 3: Median query time for random queries on a 50K document subset for the CS representation (a) and the R&CS representation (b), varying the shape and size of the compression block. Each line describes the performance for a given "compression factor." The X-axis denotes the width of the compressed block (number of words); the height of the block (number of documents) is whatever is required to maintain the compression factor. (a) There is no significant difference or identifiable trend in query run-time for different compression factors. (b) Lower compression factor yields better query run-time performance. Wider compressed blocks (fewer documents and more words) perform better than narrower but taller blocks.



(a) CS Query Performance (b) R&CS Query Performance

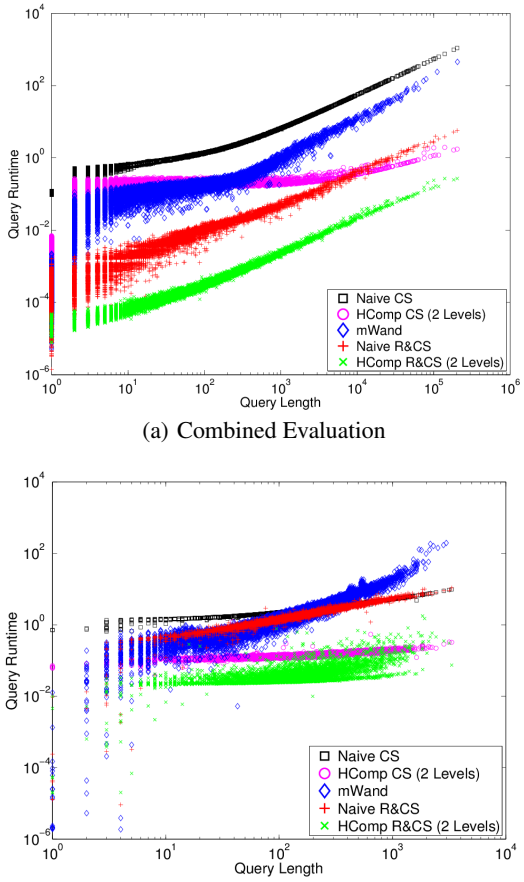
Figure 4: Query runtime for each of the 10K queries, varying the number of compression levels using the CS (a) matrix representation and the R&CS (b) matrix representation. *Naive CS* and *Naive R&CS* corresponds to the naive top-K strategy which uses no compression. Note the log-log scale.

In summary, R&CS representation is typically significantly more efficient at the cost of a 2x increased memory consumption. HComp provides between *1-3 orders of magnitude* of *additional* performance gain depending on query length and matrix representation. The performance figures in Fig. 5(a) demonstrates an incredible *4 orders of magnitude* of performance differences between the fastest and the slowest algorithms, with HComp R&CS consistently being the fastest algorithm.

Finally, to demonstrate the performance of the HComp algorithm, we evaluate on 10,000 randomly generated word queries on the full Wikipedia dataset comprising of 2,312,594 documents and 7,574,761 unique words. The median top-10 query time is **25** μ**s**, and 95% of all queries complete within **200** μ**s**.

4.2 Query Memory Utilization

A concern with large datasets is the amount of memory required to complete a given query. For instance, implementations based directly off of the algorithm template in Alg. 1 will require $O(W)$ memory per query which can be extremely large for large datasets. The HComp algorithm compacts the heap size by letting each heap element represent the equivalent of a range of words, expanding the range only when necessary. We demonstrate the effectiveness of this



(a) Combined Evaluation

(b) Combined Inverted Evaluation

Figure 5: (a) A combined comparison of all top-K strategies for both matrix representation strategies and for the mWand algorithm. (b) A combined comparison of all top-K strategies for the transposed problem (retrieving most similar documents) for both matrix representation strategies and for the mWand algorithm.

strategy by evaluating the maximum size of the heap encountered during the execution of the HComp algorithm using the R&CS matrix representation with two levels of compression. We observe that while the entire dataset comprises of 3.5M words, the heap never exceeds a capacity of 4807 elements.

4.3 Transposed Problem

To explain the large performance gap between the Naive CS and R&CS methods in Fig. 5(a), we perform the complete evaluate once again on the transposed problem: to find the best documents which matches a given query document. In other words, where D is the document-word matrix, and d is a query comprising of a sparse row vector of words, the task is to return the top 10 entries of Dd^T. We generate a synthetic query set by extracting a set of 5000 random documents from D. The statistics of the query set are in Table 5.

In comparison to the original problem, the transposed query lengths are much more uniform with a lower standard deviation. The largest query only has $3,330$ terms as compared to $203,248$ in the original task. We run all the implemented algorithms on the transposed problem and plot the results in Fig. 5(b). We observe that for shorter queries, mWand, naive R&CS, and naive CS all have very similar runtimes. For long queries, we observe that mWand does not scale well with query length. An important observation is

Number of Queries	5,000
Mean Query Length	212.74
Stddev. Query Length	249.79
Median Query Length	142
Largest Query Length	3,330

Table 5: Transposed Query Statistics. Query Length is the number of non-zero entries in the query vector.

that the huge 2 orders of magnitude performance gap between naive CS and R&CS no longer exists in the transposed problem; both techniques perform comparably, with mWand having an advantage for short queries. Both HComp CS and HComp R&CS still demonstrate about a 1-2 order of magnitude performance gain beyond the naive techniques with HComp R&CS being the fastest.

5. DISCUSSION

5.1 Related Work

Given the importance of applications such as information retrieval and online advertising, top-K similarity queries over document or word indexes have been a popular research topic. Here we provide an overview of existing methods and how we relate to them.

DAAT / TAAT: The most directly comparable prior methods are two families of algorithms known as *document-at-a-time* (DAAT) and *term-at-a-time* (TAAT) from the IR community. [4, 10, 27, 11] This line of work focuses on computing the K most similar documents to a query document (i.e., the transpose of the word similarity problem). DAAT and TAAT strategies differ by their search orientation: DAAT approaches score one document at a time, skipping documents that are bounded away from being a top-K contender; TAAT approaches process one term at a time while accumulating similarity score for multiple candidate documents. The main design assumption for these approaches is that the indexes (representations of the document-word matrix) reside on disk; hence these methods are optimized for minimizing disk seeks and reads. Subsequent research improves these methods for in-memory indexes [16], but still relies on the same procedure. The Wand upper bounding scheme can also be accelerated using a block-max technique [15] which shares similarities with the compression scheme described in this paper.

Our approach differs in that it is designed with the assumption that both the index and the inverted index can fit into memory, and that query lengths can be arbitrarily long. We consider the in-memory assumption to be reasonable given that terabyte-memory machines as well as cluster setups are becoming increasingly affordable and commonplace.

Most importantly, these techniques typically assume relatively short queries of 10s of terms and do not scale well with query length. The need to manage extremely large query lengths is a fundamental necessity for k-NN inner product search.

Space Partitioning Trees: Space partitioning techniques such as KD-Trees [6] and M-Trees [12] provide fast exact K-nearest neighbor retrieval [7, 24] in settings where candidates can be represented as a point in an Euclidean or metric space. These techniques depend on the triangle inequality to provide upper bounds on candidates and are thus not easily extended to the top-K inner product setting. [5]

Approximate Methods: Approximate counting and approximate top-K methods are widely used [13, 19, 17, 10, 3, 18, 21] and can accelerate the speed of data retrieval dramatically. Many DAAT/TAAT procedures also include the ability to provide approximate solutions for lower computational effort. Our technique in this paper however targets the exact top-K retrieval problem and approximations are outside the scope of this work. However, some possible modifications to the algorithm to permit approximations is briefly discussed in Sec. 6.

Locality Sensitive Hashing: Locality Sensitive Hashing (LSH) covers a broad class of approximate techniques to solve this problem [1, 23], including situations where alternate similarity metrics are used (such as L_2 distance or Jaccard coefficient among many others). For instance, the use of random projections [14, 20] has strong approximation guarantees on both distance and inner products, making it suitable for both K-nearest neighbor and top K inner product settings. Our approach, on the other hand, provides a solution to the exact top-K retrieval problem only in the inner product setting, In particular, since we do not introduce additional approximations, our technique can be stacked on top of inner product LSH techniques, allowing them to be accelerated even further. Finally, while the technique described in this paper shares similarities with random projection methods or hashing methods [25], one significant difference is that we do more than dimensionality reduction on individual candidate vectors (words or documents). Instead we operate on the entire matrix, permitting several candidate vectors to be combined and evaluated simultaneously, accelerating query performance significantly.

Other Relations: The HComp algorithm subsumes a number of other different filtering strategies one might use for the top-K problem. For instance, using $L^c_{1,\infty}$ norm for matrix compression is similar to the creation of a count-min sketch [13] with a sketch size of 1. The key difference is that the count-min sketch uses a hash function for "compression", while the HComp algorithm described in this paper uses a blocking scheme. Our analysis of the HComp algorithm extends to the hashing case. Furthermore, the HComp algorithm can be extended in the same way as count-min sketch to multiple sketches by generating multiple compressed matrices using different word orderings / hash functions or even compression factors.

In the binary matrix/query case, if the $L^c_{\infty,\infty}$ norm is used for compression, we obtain the equivalent of a two dimensional Bloom Filter [9] with one hash function. The HComp procedure can be interpreted as using the Bloom Filter to quickly test for query intersection before committing to a costly inner product operation.

Finally, using compression block sizes of (N documents $\times$ 1 word) where N is the total number of documents in the dataset will recover the simple upper bounding method in Sec. 2.2. For instance, picking $p = 1$ for the `CompressMatrix` procedure in this setting will result in the algorithm evaluating in order of the 'most common word first".

5.2 Parallelization

Direct parallelization of the HComp algorithm is non-trivial due to the use of a central heap. While evaluation of heap elements can be parallelized, the heap can becomes a central source of contention.

On the other hand, just like any other exact top-K algorithm, a simple parallelization scheme can be used where the document-word matrix is sliced into long vertical pieces. Each thread owns a vertical slice of matrix and thus a disjoint set of words. To perform a top-K query, the query is broadcast to all threads where each thread computes and returns the top-K results from its own disjoint set of words. The responses are pooled together, and the top-K returned. This procedure can be also distributed easily.

[5] Exceptions are situations where all candidate vectors are normalized and thus for a query v, $\arg\min_d (d - v)^2 = \arg\min |d|^2 - 2d^T v + |v|^2 = \arg\max d^T v$.

6. CONCLUSION AND FUTURE WORK

In this paper, we present a novel algorithm for computing top-K inner product similarity statistics called HComp which works by constructing a hierarchy of smaller compressed matrices, using Hölder style matrix inequalities to provide bounds on the larger matrix. The matrix compression scheme can be incrementally updated, permiting streaming/online insertion of new documents.

We demonstrate that the HComp algorithm can provide 1-3 orders of magnitude performance gains as compared to mWand and naive methods while requiring only a small increase in memory footprint. Furthermore, query-time memory utilization is extremely small, empirically requiring a heap size of only thousands for a vocabulary size in the millions. Finally, HComp scales well from short queries to extremely long queries with over 100,000 terms.

Potential future work includes an extension of the scheme to support disk-based document storage. In particular, the hierarchical matrix compression scheme could permit a set of compressed matrices to be stored in memory, allowing high speed, high-quality bounds to be obtained at the level of query tiles, and going to disk only when an exact result needs to be obtained. Furthermore, as discussed in Sec. 3, the only requirement for the matrix representation is the ability to compute inner products of a query vector against a contiguous group of the columns in the matrix. This requirement can be easily satisfied with many disk based matrix representations. Alternatively, the upper bounding technique in this paper can be combined with other disk-based DAAT / TAAT algorithms to form hybrid algorithms, combining the strength of a few different upper bounding strategies.

Next, the HComp algorithm can be easily extended to the exact top-K all-pairs inner-product problem [5, 31, 30], i.e., to find the top-K pairs of words which co-occur with each other most frequently. The key observation is that inner products between columns in the compressed matrix upper bound inner products of any pair of corresponding child columns in the full matrix. Therefore, only simple modifications to Alg. 4 are needed: use heap elements comprising of both a query-range and a candidate-range.

Finally, the Hcomp algorithm is also quite amenable to approximation strategies. For instance, the compression procedure could be "lossy," dropping low frequency counts and thus improving sparsity of the compressed matrices. Alternatively, the scoring of single word scores could be increased by a constant factor $\eta \geq 1$ thus allowing single word ranges to rise up faster in the heap. This has the benefit of providing a bounded approximation where lowest scored entry of the approximate top-K is at most a factor of $\frac{1}{\eta}$ away from the the lowest scored entry of the exact top-K.

7. REFERENCES

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51:117–122, Jan. 2008.

[2] P. M. Q. A. André F. T. Martins, Mário A. T. Figueiredo. Kernels and similarity measures for text classification. In *Proceedings of the 6th Conference on Telecommunications*, 2007.

[3] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *SIGIR*, pages 372–379, 2006.

[4] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: index-access optimized top-k query processing. In *PVLDB*, pages 475–486, 2006.

[5] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, September 1975.

[7] G. Beskales, M. A. Soliman, and I. F. IIyas. Efficient search for the top-k probable nearest neighbors in uncertain databases. *PVLDB*, 1:326–339, August 2008.

[8] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag, Secaucus, NJ, USA, 2006.

[9] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970.

[10] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, pages 426–434, 2003.

[11] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *SIGIR*, pages 97–110, 1985.

[12] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *PVLDB*, pages 426–435, 1997.

[13] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, April 2005.

[14] S. Dasgupta and A. Gupta. An elementary proof of a theorem of johnson and lindenstrauss. *Random Struct. Algorithms*, 22:60–65, January 2003.

[15] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *SIGIR*, 2011.

[16] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Y. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. In *PVLDB*, 2011.

[17] A. Goyal, J. Jagarlamudi, H. Daumé, III, and S. Venkatasubramanian. Sketching techniques for large scale nlp. In *Proceedings of the NAACL HLT 2010 Sixth Web as Corpus Workshop*, WAC-6 '10, pages 17–25. Association for Computational Linguistics, 2010.

[18] N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In *WISE*, volume 3806, pages 470–477, November 2005.

[19] P. Li and K. W. Church. A sketch algorithm for estimating two-way and multi-way associations. *Comput. Linguist.*, 33:305–354, Sept. 2007.

[20] P. Li, T. J. Hastie, and K. W. Church. Very sparse random projections. In *KDD*, pages 287–296, 2006.

[21] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. *PVLDB*, 29:129–140, 2003.

[22] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14, October 1996.

[23] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *Proc. VLDB Endow.*, 5(5):430–441, Jan. 2012.

[24] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD*, 1998.

[25] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, A. Strehl, and V. Vishwanathan. Hash kernels. In *AISTATS*, April 2009.

[26] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, and S. Vishwanathan. Hash kernels for structured data. *JMLR*, 2009.

[27] H. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Information Processing & Management*, 31(6):831 – 850, 1995.

[28] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt. Graph kernels. *JMLR*, 2010.

[29] X. Wang, A. Smalter, J. Huan, and G. H. Lushington. G-hash: towards fast kernel-based similarity search in large graph databases. In *EDBT*, 2009.

[30] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, March 2009.

[31] J. Zhai, Y. Lou, and J. Gehrke. Atlas: a probabilistic algorithm for high dimensional similarity search. In *COMAD*, pages 997–1008, 2011.