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 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.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures; H.3.3 [**Information Systems**]: Information Search and Retrieval—*Search Process*

Keywords

Top K, Inner Product, Nearest Neighbor

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 [4]. 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 [3] (among many others) have been proposed, which provide fast exact K-nearest neighbor retrieval.

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 [13, 12]. Sub-string kernels can be used for document analysis [2] or biological sequence analysis. In these cases, the hash kernel method described in [10] can be used to construct an explicit representation of the feature space thus mapping the task into the top-K inner product regime.

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 [7, 5, 9, 11, 6], 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.

Our running example in this paper focuses on finding the *exact top-K* largest inner products given a query word. 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 to provide a much wider range of query lengths, allowing us to benchmark the system on both extremely short and extremely long queries. We define the similarity function between words w_1 and w_2 as the inner product of the two column vectors w_1 and w_2:

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

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 $v^T D$.

*This work was done while interning at Microsoft Research.

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.

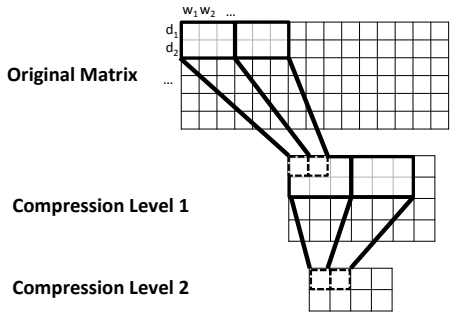


Figure 1: A document collection is represented as a sparse matrix containing arbitrary non-negative values (counts / tf-idf scores / etc). The matrix is then compressed into a smaller matrix were each element in the smaller matrix represents a block in the original matrix.

We focus strictly on the in-memory setting, 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. 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. [1]

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.
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.
4. The procedure must scale well with query length.

Our solution, which satisfies all key conditions of efficiency, involves constructing a "compressed" auxiliary matrix containing upper bounds of blocks of the 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.

[1]A long version of this paper is published as a technical report[8].

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 $w \in W$ **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**
    `// Get the true inner product value`
    Insert $(w, v^T w)$ into H
  else
    Insert w into ReturnWords
    if $|$*ReturnWords*$| = K$ **then return** *ReturnWords*
return *ReturnWords*

2.1 Hölder Compression

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.

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).

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 $\alpha \geq 1, \beta \geq 1$,

$$L^c_{\alpha,\beta}(A) = \left(\sum_j (\sum_i a_{ij}^\alpha)^{\beta/\alpha} \right)^{1/\beta}. \tag{2.2}$$

Essentially, $L^c_{\alpha,\beta}(A)$ computes the α-norm of each column, then computes the β-norm of the result footnoteEven though these definitions require $\alpha, \beta < \infty$, analogous forms for the ∞-norm (max-norm) can be defined..

Then, the fundamental result:

THEOREM 2.1. *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:*

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

The proof for this theorem as well as the choice of $L^c_{q,\infty}$ (fixing $\beta = \infty$) can be found in [8].

Essentially Theorem 2.1 allows us to compress the entire matrix A to yield an upper bound. Clearly, the quality of the bound degrades when when the matrix is larger, resulting in a trade-off between bound quality and computation efficiency. Furthermore, the procedure permits multiple levels of compression, allowing the compressed matrix can be further compressed into an even smaller matrix while retaining the upper bounding property. The following corollary summarizes the results of this section.

COROLLARY 2.2. *Given query vector v and document-word matrix D, let $\tilde{v}$ denote the p-compression of the vector v, and $\tilde{D}$*

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

Input: D : matrix to compress
Input: lvls : Number of 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'
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 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)*

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.

2.2 HComp Algorithm

We now provide a complete definition of the complete HComp algorithm. 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 compression hierarchy.

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, 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.

We observe that to implement the HComp algorithm requires only a matrix representation with the ability to compute $v^T D^{(\mathbf{c})}$ where $D^{(\mathbf{c})}$ is a contiguous range of columns in D. In the next section, we explore two different in-memory representations with this property.

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

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 $([i, \text{lvls}], \text{FirstBound}_i)$ 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*

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 [8] that suggests that the $(1, \infty)$ pair is optimal for binary data.

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. We will use the acronym **CS** to identify algorithms implemented with the Jagged Column Store.

Jagged Row And Column Store: In this representation, the document-word matrix is represented in both column format and row format. The row representation essentially acts as an inverted index. 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 post-processing pass.

3.3 Algorithms

We implement the following procedures:

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. To computing $v^T D$ in the CS representation we simply eval-

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

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 1: Wikipedia Dataset Statistics

uate inner products of v against every column, while In the R&CS representation we use the reverse index to iterate through documents containing the query word accumulating a weighted sum.

mWand: We also implement the Wand algorithm described in Broder et al. [5] with the in-memory mWand optimization in Fontoura et al. [7]. While not designed for long query lengths, it provides a reliable evaluation baseline. The Wand Iterator uses the inverted index and uses a clever upper bounding strategy to skip and ignore some of entries in the inverted index. For the purposes of this evaluation, we configure Wand to provide exact top-K.

HComp Top-K Algorithm: As noted in Sec. 2.2, the HComp algorithm only requires the matrix representation to provide the ability compute inner products against contiguous ranges of columns in D. While this is trivially provided by the CS representation, this restriction operation in R&CS 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.

Compared to the dataset evaluated in Fontoura et al. [7], which has 454M non-zero elements, the Wikipedia dataset is roughly half the size. However, since we are solving the transposed problem, the problem statistics are significantly different. In particular, we have a smaller average number of unique terms per candidate (35.35 vs 130.33), but with a much larger standard deviation (1011.02 vs 103.86); our task has much larger variation in candidate sizes.

We generate a query set by extracting 10,000 random columns from the matrix. The statistics of the query set are shown in Table 1. Most notably, we observe that we have queries lengths ranging from 1 term to over 200,000 terms. Our evaluation task is to return the exact top-10 other words co-occuring with the words in the query set. We compare against naive strategies for both CS and R&CS matrix representations as well as the mWand procedure.

4.1 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 (using only one compression level).

In Fig. 2(a) and Fig. 2(b), we plot the median runtime of varying both compression block size and compression block shape, respectively for CS and R&CS representations. We observe from Fig. 2(a) that the choice of compression factor does not impact performance under the CS representation). However, the R&CS representation is *significantly* faster with smaller compression block sizes and wider block shapes (more words than documents). Smaller block sizes improve performance by improving the quality of the upper bound, while wider block sizes decreases the number of binary searches required by maximizing the number of expanded words in each search (each restriction operation incurs a binary search. See Sec. 3.3).

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

4.2 Performance

For both matrix representation strategies, we evaluate the effectiveness of using hierarchical compression. Fig. 2(c) plots the query performance as the number of compression levels are increased when the CS matrix representation is used. Fig. 2(d) 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.

Finally, in Fig. 3(a) we plot the combined performance of all matrix representations for all algorithms: Naive, HComp and mWand. We summarize our observations here:

mWand vs Naive CS: We observe that the mWand algorithm is faster than Naive CS for short queries, but when the query length exceeds 1000, the overhead of the mWand algorithm starts to become evident and ends up slower than the Naive algorithm. For short queries (less than 100), mWand does provide about a factor of 4 speedup over the naive CS implementation which is inline with the performance gains observed in Fontoura et al. [7].

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.

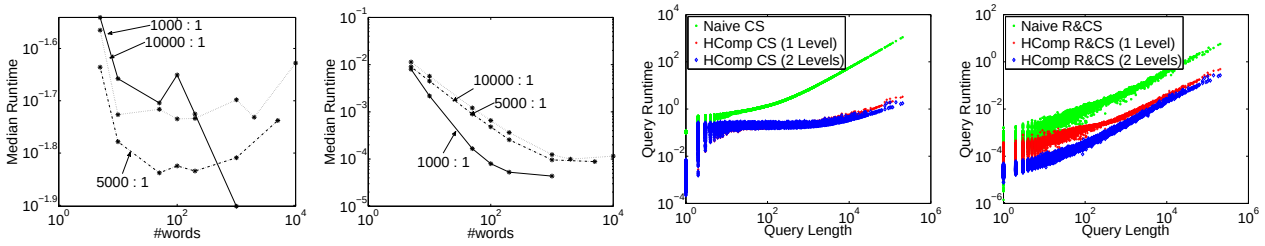
HComp R&CS vs Naive R&CS: HComp provides a consistent 1 order of magnitude performance gain over all query lengths above the Naive method using the R&CS representation. This gain is consistent on the transposed problem (see [8]).

In summary, HComp provides between *1-3 orders of magnitude* performance gain depending on query length and matrix representation. The performance figures in Fig. 3(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**.

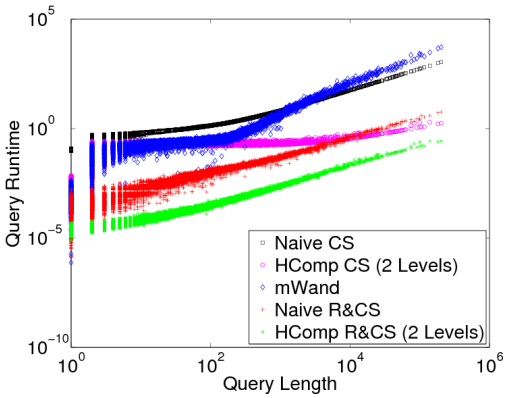
4.3 Query Memory Utilization

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. 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. This strategy is extremely effective in practice:



(a) CS Varying Shape (b) R&CS Varying Shape (c) CS Query Performance (d) R&CS Query Performance

Figure 2: ***(a,b):*** **Median query time for random queries on a 50K document subset for CS (a) and 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 is the number of documents required to maintain the compression factor.** ***(c,d):*** **Query runtime for all 10K queries, varying the number of compression levels using the CS (c) and the R&CS (d) representation. Naive CS and Naive R&CS corresponds to the naive top-K strategy. Note the log-log scale.**



(a) Combined Evaluation

Figure 3: (a) A combined comparison of all top-K strategies for both matrix representation strategies and for the mWand algorithm. Note the logarithmic axes.

we observe that using the R&CS representation with two levels of compression on the complete dataset, the heap never exceeds a capacity of 4807 elements on all evaluated queries.

5. RELATED WORK

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). 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 provide fast exact K-nearest neighbor retrieval in settings where candidates can be represented as a point in an Euclidean or metric space. These typically depend on the triangle inequality and do not easily extend to the top-K inner product setting.

Locality Sensitive Hashing: Locality Sensitive Hashing covers a broad class of approximate techniques to solve this problem [1] with strong approximation guarantees on both distance and inner products. Our approach solves the exact top-K retrieval problem.

6. CONCLUSION

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.

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

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

[5] 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.

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

[7] 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.

[8] Y. Low and A. X. Zheng. Fast top-k similarity queries via matrix compression. Technical Report MSR-TR-2012-81, Microsoft Research, 2012.

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

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

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

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

[13] 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.