Efficient Set Intersection Counting Algorithm for Text Similarity Measures

Preethi Lahoti* Patrick K. Nicholson† Bilyana Taneva†

Abstract

Set intersection counting appears as a subroutine in many techniques used in natural language processing, in which similarity is often measured as a function of document cooccurence counts between pairs of noun phrases or entities. Such techniques include clustering of text phrases and named entities, topic labeling, entity disambiguation, sentiment analysis, and search for synonyms.

These techniques can have real-time constraints that require very fast computation of thousands of set intersection counting queries with little space overhead and minimal error. On one hand, while sketching techniques for approximate intersection counting exist and have very fast query time, many have issues with accuracy, especially for pairs of lists that have low Jaccard similarity. On the other hand, space-efficient computation of *exact* intersection sizes is particularly challenging in real-time.

In this paper, we show how an efficient space-time trade-off can be achieved for exact set intersection counting, by combining state-of-the-art algorithms with precomputation and judicious use of compression. In addition, we show that the performance can be further improved by combining the best aspects of these algorithms. We present experimental evidence that real-time computation of exact intersection sizes is feasible with low memory overhead: we improve the mean query time of baseline approaches by over a factor of 100 using a data structure that takes merely twice the size of an inverted index. Overall, in our experiments, we achieve running times within the same order of magnitude as well-known approximation techniques.

1 Introduction

In recent decades, research into natural language processing (NLP) has been driven by numerous applications that rely on techniques such as clustering of text phrases [31], topic labeling [33, 25], entity disambiguation [24], sentiment analysis [42], and synonym finding [41]. Successful methods for solving these problems typically rely on the computation of some measure that determines the similarity between pairs of text phrases or named entities. Such *similarity measures* include pointwise mutual information (PMI) [9] (including variants like enhanced mutual information [45]), normalized Google distance [10], and Jaccard similarity. For example, the first-order relevance model of Mei et al. [33] generates topic labels by computing the PMI between all words or noun phrases in an input text and a set of candidate topic labels. The set of candidate topic labels can be large (e.g., more than 1000), and thus computing all PMI scores is a bottleneck for this algorithm. Another example is in the case of clustering Wikipedia articles, where a graph-based distance can be used (or considered) to measure overlap between the sets of hyperlinks embedded in articles. Computing the entire distance matrix in such applications requires evaluating a quadratic number of similarity scores.

For these techniques, similarity scores are computed for many pairs of phrases or entities. Typically, these scores are computed based on a reference document corpus, and boil down to computing three numbers: the number of documents in the corpus that contain both phrases, and the number containing each phrase individually. On a more abstract level, supporting such queries is an instance of the *set intersection counting* problem, in which we are given a collection of sets $\mathcal{S} = \{S_1, ..., S_N\}$ as input, where $S_i \subseteq [1, U]$, and asked to preprocess them in a data structure so that queries regarding the size of the intersection of any pair of sets can be answered efficiently. In the context of text similarity, S_i can be thought of as the set of document IDs of the documents containing phrase p_i. The query asks, given two indices i and j, to compute $|S_i \cap S_j|$. In this paper we focus on the case in which we receive a batch of phrases $p_1, ..., p_k$ and are asked to compute all *pairwise* intersection sizes. However, it is important to note that all techniques presented in this paper can be used for individual set intersection counting queries as well.

For NLP applications with real-time constraints, such as determining the personalized urgency and im-

*Aalto University, Espoo, Finland
†Nokia Bell Labs, Dublin, Ireland

portance of an incoming message to a user, computing similarity measures for pairs of phrases or entities in a new document must be very efficient. A single email or a blog post can have dozens of phrases. If similarity measures are to be computed for all pairs of phrases, then this can result in several thousand set intersection counting queries. Thus, for such systems, the average individual queries should take microseconds rather than milliseconds. In addition, such systems often must deal with short messages, which contain few phrases. Significant error in computing similarity scores between those phrases can have an impact on the overall quality of the application. Thus, in some situations it can be important that the system does not introduce any additional error due to approximation. Lastly, the sheer number of documents and text phrases that a system needs to store is often quite high. To accommodate real-time constraints, we require that the data structure fits in main memory. Therefore, memory consumption is an important constraint for these systems. To summarize: for such systems, data structures should support *very fast and exact queries with little space overhead*.

In this paper, we study how using varying degrees of *precomputation* affects the time-space trade-off for exact set intersection counting algorithms. During precomputation, we compute the size of the intersection between all pairs of large sets in the collection: a large set is one that exceeds a specific length threshold. We store the results of the precomputation in a large matrix, which is then compressed. We perform experiments comparing eight state-of-the-art algorithms for set intersection counting. Finally, we develop a strategy, called `ULTIMATE`, that combines the best aspects of various algorithms.

We find that precomputation is highly effective at alleviating the cost of the most expensive input queries, and can improve the mean query time for a batch of queries by an order of magnitude. Moreover, we show that the precomputed matrix of solutions has very low entropy. Therefore, when compared to baseline solutions, if precomputation is combined with *compression techniques* that support direct access, it results in only a modest increase in space and negligible increase in time. To the best of our knowledge, we are the first to study the combination of using precomputation and compression for set intersection counting.

Quite interestingly, we also find that the ranking of algorithms changes significantly when they are combined with precomputation. While standard linear merging algorithms perform best without precomputation, we find they become an order of magnitude worse than algorithms that take advantage of highly skewed ratios between the input query sets when precomputation is used. By combining precomputation and compression with such algorithms, we show that we can further improve the mean query time by over a factor of 100, with only a modest increase in memory footprint.

We perform extensive experiments on a variety of data sets. Overall, in our experiments, mean running times are under a microsecond, which is close to approximation techniques. For the scenarios described previously (computing similarity for all-pairs of phrases or entities in short documents), this is sufficient for real-time processing.

2 Related Work

Numerous algorithms were proposed for set intersection counting and *reporting*: i.e., the problem of outputting the intersection, rather than just computing its size. Since we are interested in average query time of microseconds, we consider only in-memory algorithms. Given a query on sets S_i and S_j, $s = \min\{|S_i|, |S_j|\}$ denotes the smaller of the two sets, and $\ell = \max\{|S_i|, |S_j|\}$ denotes the larger set. In our work, all data structures occupy space linear in the total size of the sets, unless otherwise stated, and all theoretical results operate in the word-RAM model of computation with word size $w = \Theta(\log \max\{U, N\})$. We consider only sequential algorithms, but note that set intersection counting problems can be parallelized via inter-query multicore parallelism [39] and distributed Map-Reduce frameworks [13]. Thus, parallelism is orthogonal to our interest: improvements to the sequential algorithm lead to improvements in the parallel setting.

Exact Algorithms: Any algorithm for set intersection reporting can be used to solve the counting problem, and since we can perform a *linear merge* on the sorted lists S_i and S_j in time $\mathcal{O}(s + \ell)$, we can support counting in the same time bound. Many software libraries, such as the `C++` standard template library (STL) provide such implementations [1] which can be easily adapted for counting. Other *binary search based merging* schemes on sorted lists [27, 14] achieve bounds of $\mathcal{O}(s(1 + \lg \frac{\ell}{s}))$ time. This is done by representing sets as sorted lists of integers, and (carefully) locating each element in the smaller list within the larger list [3]. Several experimental studies have been performed comparing variants of these type of merging algorithms [1, 2, 15].

Merging algorithms exploiting *word-level parallelism* (i.e., algorithms parameterized in terms of the number of bits, w, in a machine word) have been proposed for solving set intersection reporting. Bille et al. [4] described an algorithm taking time

[1] See `http://en.cppreference.com/w/cpp/algorithm/set_intersection`.

$\mathcal{O}(\frac{(s+\ell)\log^2 w}{w} + t)$, where t is the size of the intersection, in the worst case. Ding and König [17] gave an algorithm that takes time $\mathcal{O}(\frac{s+\ell}{\sqrt{w}} + t)$ pointing out that w is typically very small in practice (32 or 64 bits), and that $\frac{\log^2 w}{w} > \frac{1}{\sqrt{w}}$ when $w < 2^{16}$. Recently, another approach based on word-level parallelism was presented by Rao et al. [35]. They propose a simple partitioning strategy that can be applied to both linear merging and binary search based merging. It is also possible to achieve time $\mathcal{O}(s)$ by storing the sets in any *linear space perfect hashing* data structure that supports constant query time [20].

As part of our framework, we consider the STL implementation of set intersection, binary and galloping search, the search algorithm proposed by Baeza-Yates [1], the approaches by Rao et al. [35] and by Ding and König [17], as well as a hashing approach based on quadratic probing.

Precomputations and Compression: A different approach based on precomputation was proposed by Cohen and Porat [11], following a line of research from other earlier studies [44, 16]. They show how to solve set intersection counting in $\mathcal{O}(\sqrt{n})$ time, where $n = \sum_{i=1}^{N} |S_i|$, i.e., the total number of elements in all sets. Their algorithm is based on the simple observation that at most $\lceil\sqrt{n}\rceil$ sets can have size larger than $\lceil\sqrt{n}\rceil$. If we precompute the answers for all pairs of sets in which both sets exceed this length threshold, then a query for two arbitrary sets S_i and S_j can be answered in $\mathcal{O}(\sqrt{n})$ time (via hashing) if one of the sets is not large, and in $\mathcal{O}(1)$ otherwise. We adopt a parameterized version of this approach and precompute the intersection size of all pairs of sets exceeding various length thresholds.

Similar ideas have been applied to other, more general, database problem domains as well [38]: in addition to set intersection, the work focuses on constraints based on the range of the results.

Several methods have been proposed for reducing the size of the inverted index without significantly impacting running time [36, 7]. These strategies typically use compression of posting lists, based on Elias delta codes, Rice codes, and others. Recent work by Lemire et al. [30] studied vectorization-based approaches to both compression and set intersection reporting. They found that modern processors supporting single instruction multiple data (SIMD) instructions can accelerate set intersection queries. In our work, we do not consider compression of posting lists or SIMD vectorization. Instead, we store precomputed results for certain pairs of posting lists in a matrix, and compress this matrix using various compression strategies.

Approximation Algorithms: Originally designed for duplicate detection, `MINHASH` [6] is a popular technique for estimating Jaccard similarity of sets, which in turn can be used to estimate intersection sizes. The idea is to apply k different hash functions to each element in the sets and to store the minimum value for each hash function. Recently, Pagh et al. [34] showed that `MINHASH` is essentially optimal for the set intersection counting problem in terms of the space-accuracy trade-off. Since `MINHASH` achieves very fast query time, we consider this *approximate* technique to be a reasonable lower bound on the running time for our *exact* methods.

Takuma and Yanagisawa [37] proposed a hashing-based data structure, called a *cardinality filter*, that can be used to provide approximate upper bounds on intersection size in time $\mathcal{O}(\frac{s+\ell}{w})$. However, in their experiments they found that the speed gain of cardinality filters over exact set intersection counting algorithms (e.g., [17]) was limited to a factor of between 2 and 10. Finally, we note that other sketching techniques, such as count-sketch [8] and count-min [12] have been used for related counting problems in the domain of NLP [18, 19, 22, 26, 42].

3 Algorithmic Approaches

Our overall framework consists of several building blocks. We use multiple algorithms for set intersection counting, which are based on different ideas, such as merging, searching, bitwise parallelism, and hashing. Our key goal is to develop a framework that combines various algorithms together with precomputation and compression techniques in order to achieve an improved performance. Note that the framework is set up such that when testing a particular algorithm, we create only the data structures necessary for that particular algorithm. Also, note that the precomputed matrix of solutions is a separate part of our solution, and independent of the choice of intersection algorithm.

The basis of all our data structures is a *dictionary* and an *inverted index* [43, Sec. 3.2]. For a total of N distinct noun phrases in our data sets, we assign each phrase with a unique integer identifier $j \in [1, N]$. The inverted index I is an array, where $I[j]$ is the *posting list* for the phrase with identifier j: i.e., $I[j]$ is the sorted list of document IDs of the documents containing the phrase.

Linear and binary search based merging: Our first baseline algorithm (`STL`) is a slightly modified version of the code for `std::set_intersection` that merges two sorted lists. Instead of reporting the common elements, we increment a counter when an element in the intersection of the two lists is detected.

We also implemented two binary search based merging algorithms that work by initially ordering the two lists by length so that S_i is the smaller list, and S_j the larger. While there are still elements remaining in the smaller list, we perform a search in the larger list for the current element in the smaller list, maintaining an iterator for the lower bound of previous searches. We experimented with both regular binary search (`BINARY`) and galloping search (`GALLOP`). Finally, we implemented the Baeza-Yates [1] search algorithm (`BY`) based on double binary search. The idea is to search for the median element of S_i in S_j. Let the position in S_j where the search ended be k. The algorithm then recurses on two sub-problems: searching elements smaller than the median in S_i to the left of k in S_j, and elements bigger than the median to the right of k.

Two-Level Bucketing: We implemented a modification of the previous algorithms, inspired by Rao et al. [35][2], intended to exploit bitwise parallelism, though with weaker theoretical guarantees than other work [4, 17, 37]. This strategy divides the universe U into buckets of size w, and numbers the buckets with $[1, \lceil \frac{U}{w} \rceil]$. We store a posting list for all non-empty bucket numbers. For each non-empty bucket B, we also store a w-bit vector in which bit $i = 1$, if and only if the original posting list contains value $(B-1)w + i$. Thus, each *transformed posting list* has the form (b, v) where b is the bucket number, and v is a bit vector of length w (i.e., a single word). For example, if $U = 20$, $w = 4$, and $S_i = \{1, 3, 7, 13, 15, 18, 19, 20\}$, then the transformed posting list (counting bit positions from the left and starting with 1) would be $\{(1, 1010), (2, 0010), (4, 1010), (5, 0111)\}$. Using this structure, we perform a search by intersecting the two lists of bucket numbers. When two tuples $(b_p, v_p) \in S_i$ and $(b_q, v_q) \in S_j$ are identified such that $b_p = b_q$, then we increment the intersection counter by the number of bits set in the bitwise-AND of v_p and v_q. This can be done using a *POPCNT* instruction which is supported by many modern CPUs. Thus, we implement two strategies, `BSTL` and `BIT`, for which we use the algorithms `STL` and `GALLOP`, respectively, to intersect the bucket numbers in this structure.

Hashing-based approaches: We implement a few approaches based on pseudorandomness. The first strategy constructs and stores a hash table for each set. This allows us to directly lookup if an element is present in the set or not. Given a query for two sets S_i and S_j, we iterate over the smaller set S_i and lookup each element in the bigger hash table. If we use perfect hashing, the query time is $\mathcal{O}(|S_i|)$. We experimented with several implementations of hash tables and found that Google's `dense_hash_set`[3], which uses quadratic probing, had the best performance. However, we note that this speed comes at the cost of high space overhead: the hash table has only 50% occupancy. In our experiments we call this approach `GDHASH`.

We also implemented Ding and König's[17] algorithm `RanGroupScan`, which we call `FS` in our experiments. This approach is a two-level hashing-based data structure. First, it partitions the input sets into smaller subsets. Then, each small subset is represented by a second level data structure in which we use universal hash functions to map the constituent elements to bits in a constant number of machine words. In our implementation we used two hash functions per small subset. For each hash function, we used the multiply-shift scheme described in Thorup's work [40]. During a query, we iterate over all pairs of the smaller subsets of the two input sets. Since the small subsets are represented by machine words, we use bitwise-AND operations to determine if their intersection is non-empty. When a non-empty intersection is detected, the true intersection is computed by an `STL` merge of the two smaller subsets.

Ultimate Intersection Algorithm: The approaches described above are conceptually very different to each other. For example, while the hashing methods require the computation and the storage of a hash table but provide a constant lookup time, the search algorithms do not use an auxiliary storage but their running time is often slower. An ideal approach would combine the best aspects of these various algorithms to achieve better performance. In our experiments we analyze different algorithms by focusing on the set sizes or ratio between the sizes of the two sets. We then design a new algorithm that essentially chooses between `STL` and `GDHASH` at run-time. We call this approach `ULTIMATE` and provide more insights and details in Section 5.

Precomputed Matrix and Compression: We implemented a complimentary approach of precomputation, that is a parameterized version of Cohen and Porat's approach [11]. We select a value $\lambda \in [1, U]$, and posting lists with length exceeding λ are called *large*. Let γ denote the number of large lists, each of which is assigned a unique integer identifier. We store a $\gamma \times \gamma$ matrix $\mathcal{M}$, in which entry i, j stores the intersection size of the large lists with identifiers i and j. By making the matrix upper triangular (i.e., only storing entries when $i < j$), the space cost is reduced by a factor of two. Thus, by storing $\binom{\gamma}{2}$ additional integers, the theoretical

[2]Specifically, our approach is equivalent to their `bitlist-fix` strategy, which they found to be the fastest.

[3]`http://goog-sparsehash.sourceforge.net/doc/dense_hash_set.html`

Algorithm 1: Algorithm to compute matrix $\mathcal{M}$.

```
1: procedure COMPUTE-MATRIX(I, λ)
   ▷ I is an inverted index, λ is a nonnegative integer
2:     γ ← 0
3:     ϒ is a map from integers to lists
4:     for each postings list P in I do
5:         if P.length ≥ λ then
6:             γ ← γ + 1
7:             P.matrixID = γ
8:             for each doc-ID d in P.posting_list do
9:                 ϒ[d].push(γ)
10:            end for
11:        end if
12:    end for
13:    M is a γ × γ matrix of zeros
14:    for each list L in ϒ do
15:        for i in 1 to L.length do
16:            for j in i + 1 to L.length do
17:                M[L[i]][L[j]] ← M[L[i]][L[j]] + 1
18:            end for
19:        end for
20:    end for
21:    return M
22: end procedure
```

worst-case query time becomes $\mathcal{O}(\lambda)$ if combined with perfect hashing.

We also implemented a variant of this approach that applies compression to the matrix $\mathcal{M}$. The intuition, which we have verified for all our data sets, is that values in $\mathcal{M}$ are mostly small integers. For compressing the matrix, we tested the following compression libraries, in addition to the uncompressed variant (`None`):

1. Frame-of-Reference (`FOR`) [4]: This method divides the input data into blocks, and determines the minimum element of each block [21, 29]. All values in a block are normalized relative to the minimum element, and we determine the *maximum* number of bits b required to store each normalized value. The output is the minimum element concatenated with b and the sequence of normalized values. This allows direct access through arithmetic and bit-masking.
2. Directly Addressable Codes (`DAC`) [5]: This method is essentially an optimized variable length encoding scheme that uses succinct data structures to provide direct access. We use the most space efficient version of the code[5] from [5] that has no restrictions on the number of levels.
3. Google's `snappy` library (`SNAPPY`) [6]: For an additional baseline, we implement an LZ77-based compressor and use the following strategy: divide rows of the matrix into blocks of length 128 integers[7], then compress each individual block. When an access is performed, we decode the block and read the integer. Note that if an access is made to the same block again, we read the already uncompressed data, rather than decompress the same block again.

Algorithm for Batch Precomputation: Finally, since entries of the matrix store the results of the most time consuming set intersection queries, we compute them in batch using auxiliary data structures, rather than one-at-a-time. Algorithm 1 is a straightforward approach that solves this batch problem in time $\mathcal{O}(\gamma^2 + \sum_{i,j} \mathcal{M}_{i,j})$. The basic idea is to select the posting lists from the inverted index that exceed the threshold λ. We then revert these lists, creating a mapping from the individual document IDs to these frequent phrases. At this point the algorithm can traverse each list with two nested loops, incrementing the intersection counts of each pair of phrases contained in each document. For our data sets this led to a preprocessing time measured in minutes, rather than hours (in the case where all pairwise intersections are computed independently).

4 Experimental Setup

Data Sets: To objectively compare different approaches for set intersection counting in the context of NLP, we use the following publicly available real-world data sets:

1. `Enron`: A collection of public emails from the Enron corporation.[8] We composed documents using the email subject and body. During parsing, we removed the forwarded content (to avoid duplication due to email forwarding), email signatures and disclaimers.
2. `Amazon`: A collection of electronics reviews[9] scraped from Amazon by McAuley and Leskovec [32]. We created documents using the product name, review summary, and review text fields.
3. `Wikipedia`: The English language Wikipedia[10] processed with WikiExtractor.[11]

[4] `https://github.com/cruppstahl/libfor`
[5] `http://lbd.udc.es/research/DACS/`
[6] `https://code.google.com/p/snappy/`
[7] We experimented with several parameters and found 128 provides a good space-time trade-off.
[8] `https://www.cs.cmu.edu/~./enron/`.
[9] `https://snap.stanford.edu/data/web-Amazon.html`
[10] `http://dumps.wikimedia.org/enwiki/latest/` from March 7th, 2015.
[11] `http://medialab.di.unipi.it/wiki/Wikipedia_Extractor`

Figure 1: Log-log plot of total number of posting lists of length larger than x for the `Amazon` data set. This shows the trade-off between λ and γ. We omit the plots for the other data sets since they are similar.

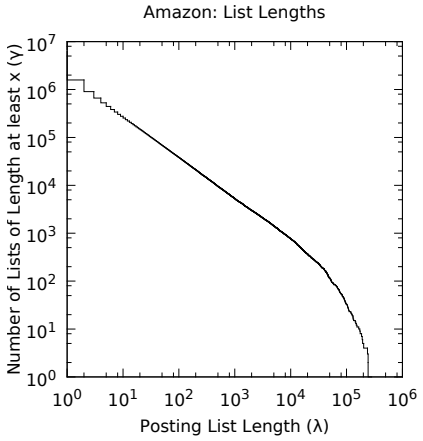
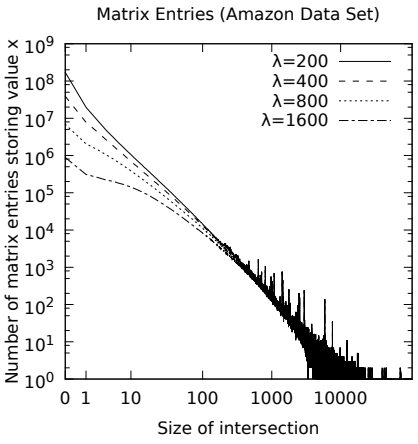


Figure 2: Log-log plot of number of entries (y-axis) in the matrix with specific value (x-axis) for `Amazon` data set.



From each document in our data sets we extracted all *noun phrases*[12] using the Natural Language Toolkit (NLTK)[13]. We represented each document as a list of its noun phrases. Note that the efficient extraction of noun phrases from a given document is orthogonal to our work.

For the data set `Wikipedia`, we created an alternate data set called `WikiDocLinks`, where instead of the noun phrases, we extracted all distinct out-links (links to other Wikipedia pages). Thus, each distinct out-link can be considered to be a phrase associated with a document.

Data Sets Statistics: Table 1 summarizes the basic statistics of our data sets. In Table 2 we examine the number of phrases (in the case of `Amazon`, `Enron`, `Wikipedia`) or entities (in the case of `WikiDocLinks`), γ, having lists longer than λ. The table shows a linear relationship between λ and γ: doubling λ reduces γ by roughly a third in most cases. Based on these data sets, we speculate that $\gamma = \mathcal{O}(N/\lambda)$ holds for many data sets used for computing similarity measures. We plot λ and γ for the `Amazon` data set in Figure 1, and find a Zipf-like distribution similar to the well-known distribution for unigrams or k-grams [23].

In Figure 2 we show the distribution of the entries in the precomputed matrices for the data set `Amazon` (the other data sets are similar), over various values of λ. The properties that can be seen in the Figure are:

1. A large percentage of the values in the matrices are numbers smaller than 10, with the most frequent number being zero.

2. Decreasing λ increases the percentage of these small numbers in the matrix.

Based on these properties, we suspect that as λ is decreased, the compressibility of the matrix should also increase.

Machines, Programming Languages, Implementation Details: All experiments were run on a Intel(R) Xeon(R) CPU E5-4657L v2 (8x CPU @ 2.40GHz) with 32 GB of RAM running Red Hat Enterprise Linux Server release 6.8. All code was written in `C++` and compiled in `g++` (Red Hat 4.4.7-16) with optimization level `-O2`. Wall clock time was measured using `clock_gettime`: `clock_getres` reported the system has 1 nanosecond resolution. For the popcount function in `BIT` and `BSTL` we used `__builtin_popcountl`, compiled with the flag `-Mpopcnt`. For our experiments, we selected 100 documents uniformly at random without replacement from each data set, and generated queries by taking all pairs of phrases in each selected document. This resulted in: $142,381$ total queries for `Amazon`, $1,050,084$ for `Enron`, $198,055$ for `WikiDocLinks`, and $1,377,778$ for `Wikipedia`.

5 Experimental Results

In this section we answer the following questions:

1. How do the implemented algorithms compare against each other in terms of running time?

[12] For simplicity we used the pattern `<NN.*>+` and extracted all nested patterns up to a maximum length of 4 words.

[13] `http://www.nltk.org/`

Table 1: Data sets statistics. Raw Data is the space in MB occupied by the data set before parsing it into phrases, Phrases/Entities is the total number of unique phrases/entities, and Total Postings is the sum of the lengths of all posting lists.

Data Set	Raw Data (MB)	Phrases/Entities	Total Postings
Amazon	859	6,227,590	61,031,347
Enron	1,435	5,399,167	65,887,789
WikiDocLinks	10,658	6,046,575	132,541,204
Wikipedia	10,658	85,219,540	1,200,639,776

Table 2: Number of phrases (γ) for which length of posting lists is larger than λ for each data set.

Data Set	$\lambda = 50$	$\lambda = 100$	$\lambda = 200$	$\lambda = 400$	$\lambda = 800$
Amazon	84,005	53,652	35,112	18,286	9,552
Enron	44,772	32,077	22,267	13,831	7,500
WikiDocLinks	29,300	18,617	11,409	6,883	4,208
Wikipedia	428,980	303,841	224,331	143,004	81,772

2. Based on our observations about this comparison, how should we define our ULTIMATE algorithm?
3. What is the effect of precomputation on the relative order of performance of algorithms?
4. What is the influence of compression on the running time and memory consumption?
5. Which settings achieve good space-time trade-off?

Comparison without Precomputation: We begin by giving an overview of each algorithm's performance in case we do not use precomputation ($\lambda = \infty$). We present boxplots of the running times of individual queries for each data set in Figure 3. Note that for the data set Wikipedia the machine did not have enough memory to run FS or GDHASH.

For comparison purposes, we also implemented MINHASH [6] with $k = 100$ universal hash functions. If a set has size $\ell < k$, we store the results of the first ℓ hash functions. This approach has well-known issues with overestimation for sets that have small Jaccard similarity (see [28] for example), but we use it primarily to compare to the running time of our exact algorithms.

We have the following insights based on these results:

1. The running time of the algorithms is highly influenced by the running time of outlier (worst-case) queries.
2. The simple baselines STL and BSTL are highly competitive when no precomputation is used, with BSTL having the best mean running time overall. Similarly, the performance of BIT tends to be better than GALLOP. This implies that the simple word-level bucketing approaches can be helpful to improve performance. As noted by Rao et al [35], reordering document IDs in the inverted index can provide further benefit by increasing the average number of set bits per transformed posting. We experimented with reordering as well and found that the order of document IDs can influence the running time between a factor of 1.2 to 2.1 for these strategies.
3. FS performs worst of all the strategies. As mentioned in the paper by Ding and König [17], this approach does well only when the ratio between the two input set sizes is close to one. However, even when restricted to this case we found that the performance was only on par with STL.
4. Finally, compared to MINHASH many of the compared algorithms are over 1000 times slower in terms of mean running time (for example on the Wikipedia data set). Thus, it is clear that, on their own, these algorithms do not achieve query times comparable to MINHASH.

Defining the ULTIMATE Algorithm: We stratified the data to examine the performance of the algorithms for different list lengths. We found that when the ratio between the larger list length and smaller list length is close to 1, BSTL or STL perform very well. However, as the ratio increases to 10 or larger, GDHASH quickly becomes the best strategy by increasing margins. Combining this observation with the observation that GDHASH uses a lot of memory, we defined ULTIMATE as follows. If the length of a list is above a threshold $t_1 \geq 1$, then we construct and store a hash table for the list. During a query, if the length of one of the lists is at least t_1 and the ratio between the sizes is at least another threshold $t_2 \geq 1$, then we use the GDHASH algorithm. Otherwise, we use STL. Thus, the parameters t_1 and t_2 interpolate between STL and GDHASH: when both

Figure 3: Comparision of algorithms across data sets for $\lambda = \infty$. Each box shows the 3 quartiles (25th percentile, median, and 75th percentile). The whiskers extend to points that lie within 1.5 inter-quartile range of the lower and upper quartile. Red square represents the mean value. For reference we plot the mean running time of MINHASH as the dashed red line.

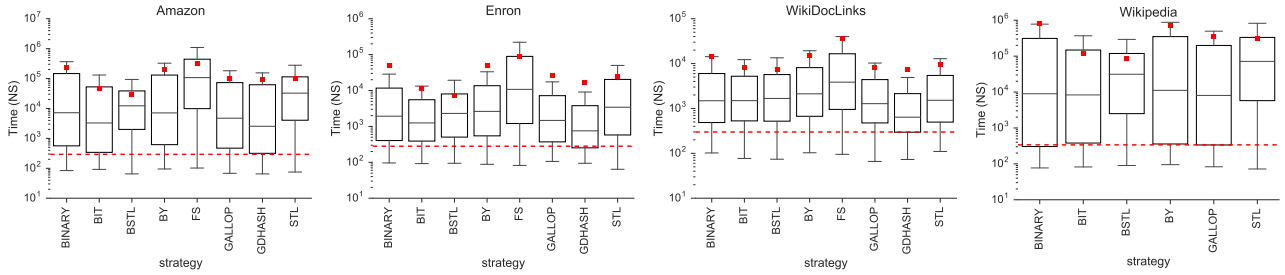
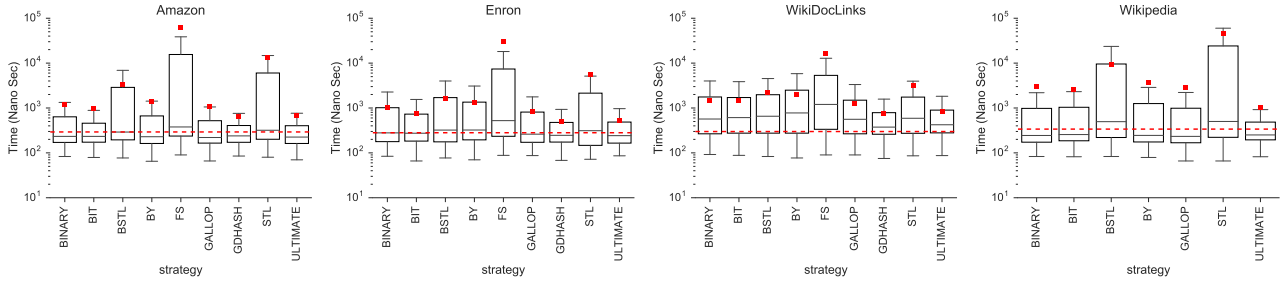


Figure 4: Comparision of algorithms for Amazon, Enron, and WikiDocLinks with $\lambda = 200$, and for Wikipedia with $\lambda = 500$. The dashed red line refers to the mean running time of MINHASH.



are infinity, then the algorithm is equivalent to STL, and when both are 1 the algorithm is equivalent to GDHASH. For our experiments we set $t_1 = 100$ and $t_2 = 10$ as we found that they provided a good trade-off between space consumption and running time.

Comparison with Precomputation: In Figures 4 and 5 we present results for all algorithms, including ULTIMATE, when they are combined with precomputation. In Figure 4 we show the results for $\lambda = 200$ for the data sets Amazon, Enron, and WikiDocLinks, and $\lambda = 500$ for Wikipedia. The reason for this difference is that Wikipedia is a much larger data set, and setting $\lambda = 200$ was prohibitive in terms of memory consumption. In Figure 5, we consider the effect of precomputation for various values of λ, including $\lambda = \infty$ (i.e., the case of no precomputation). Note that compression is not used in these experiments. We have the following observations:

1. For most algorithms, precomputation improves performance by up to several orders of magnitude. However, the ranking of algorithms without precomputation changes when precomputation is used: both STL and BSTL become the 2nd and 3rd slowest algorithms, respectively (only FS is slower). This change is especially visible in Figure 5 in the transition between $\lambda = \infty$ and $\lambda = 800$. This implies that adding precomputation not only improves performance, but that we *must* consider the effects of precomputation when evaluating such algorithms, as precomputation changes the distribution of query set sizes for the algorithms.
2. Overall, we find that, in terms of query time, GDHASH is the fastest when precomputation is used, with ULTIMATE in second, and BIT in third.
3. Finally, the gap between MINHASH and the exact algorithms is much smaller with precomputation: ULTIMATE has mean query time only between 1.9 to 3.1 times slower than MINHASH.

Comparison of Different Compression Strategies: Finally, we discuss the effect of compression on the precomputed matrix. In Figure 6 we show the memory consumption of the precomputed matrix (black) compared to storing the inverted index (gray) for STL on the data set Amazon (the plots for the other data sets are similar). We note that the relative ordering of the compression strategies remains fixed across all data sets: DAC is always the smallest achieving an average compression ratio (the uncompressed size divided by the compressed size) of 10 compared to None (i.e., no compression used), and FOR has an average

Figure 5: Performance of algorithms for varying values of λ.

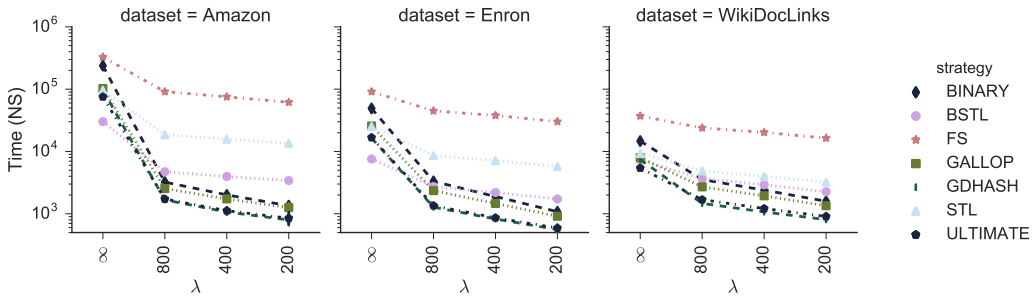


Figure 6: Impact of various compression strategies on the overall memory consumption. The values correspond to data set `Amazon`. We observe a negligible increase in space compared to the size of the inverted index when compression is used, especially for larger values of λ.

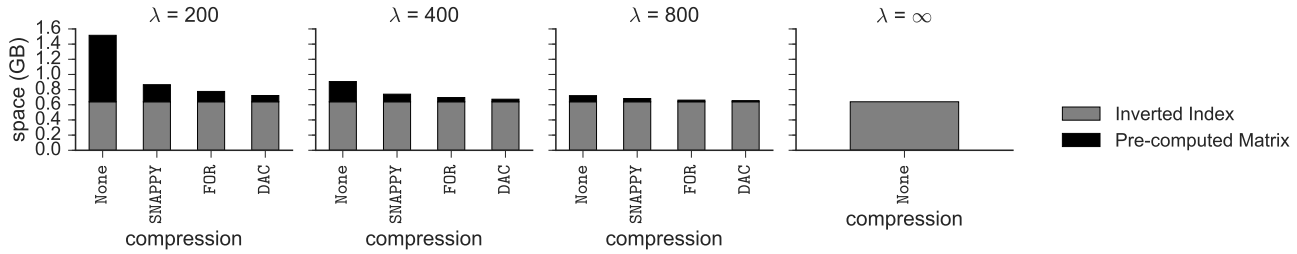
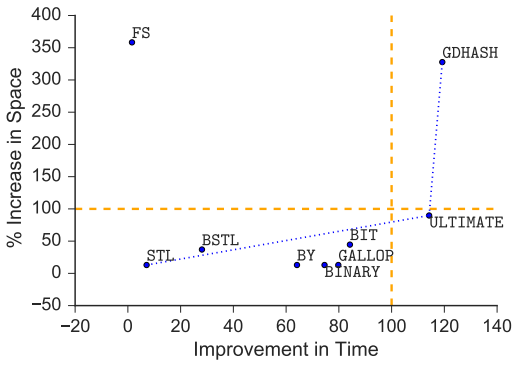


Figure 7: Space-Time trade-off for each algorithm. X-axis shows factor of improvement over `STL` in query time, Y-axis shows percentage increase in space. Figure settings: $\lambda = 200$, `DAC` and `Amazon`.



compression ratio of 5.55 across the four data sets.

In Table 3 we show the relative speedup achieved for each algorithm when combined with precomputation and the various compression algorithms. We note that `FOR` is the fastest compression algorithm for direct access, achieving speed only 2% slower on average than `None`, while `DAC` is 6% slower on average than `None`. Based on this table, we can see that all algorithms achieve a considerable speedup when combined with precomputation. For `ULTIMATE`, precomputation results in speedups of between a factor of 10 and 295 when combined with `DAC`. This comes at doubling the space of the inverted index (increasing it by at most 101 percent).

In Figure 7 we present the space-time trade-off for the various algorithms with precomputation and compression. We show `DAC` compression which achieves the highest compression ratio. We observe that the points `GDHASH`, `ULTIMATE`, and `STL` are part of a curve, as `ULTIMATE` is a parameterized approach that essentially becomes either `GDHASH` or `STL` in the extreme cases. We can see that this curve has a sharp knee in which a desirable trade-off can be found between the low space consumption of `STL` and the fast running time of the quadratic probing hash table approach of `GDHASH`. The space-time trade-off can be further explored via grid search to find the right combination.

Table 3: Values in the table represent improvement in time in terms of speedup over STL at $\lambda = \infty$, and the corresponding percentage increase in space. We observe that FOR has the best speedup, DAC has the best space, and GDHASH is the fastest strategy followed by ULTIMATE.

		Multiplicative Speedup				Percentage Increase in Space			
Data Set	**Strategy**	None	DAC	FOR	SNAPPY	None	DAC	FOR	SNAPPY
Amazon	BINARY	80.91	74.57	76.64	57.91	137.09	12.96	21.33	35.26
($\lambda = 200$)	BIT	99.84	84.19	92.61	65.56	168.72	44.60	52.97	66.89
	BSTL	29.45	28.09	28.99	26.22	161.12	36.99	45.36	59.29
	BY	70.71	64.12	67.12	52.36	137.09	12.96	21.33	35.26
	FS	1.58	1.56	1.57	1.55	482.45	358.33	366.70	380.63
	GALLOP	89.56	79.80	80.96	61.34	137.09	12.96	21.33	35.26
	GDHASH	149.10	119.22	146.53	89.54	451.83	327.70	336.08	350.00
	STL	7.23	7.10	7.16	6.96	137.09	12.96	21.33	35.26
	ULTIMATE	141.63	114.22	140.41	81.38	213.90	89.78	98.15	112.08
Enron	BINARY	24.97	23.55	24.94	20.55	276.19	33.01	58.46	85.64
($\lambda = 200$)	BIT	33.32	30.09	32.13	25.62	356.86	113.68	139.13	166.30
	BSTL	15.24	14.38	15.12	13.76	349.61	106.43	131.88	159.06
	BY	18.76	17.83	18.83	16.20	276.19	33.01	58.46	85.64
	FS	0.84	0.84	0.84	0.83	616.19	373.00	398.45	425.62
	GALLOP	29.98	27.25	29.43	24.72	276.19	33.01	58.46	85.64
	GDHASH	49.96	43.62	50.85	38.58	582.47	339.30	364.75	391.93
	STL	4.47	4.39	4.42	4.29	276.19	33.01	58.46	85.64
	ULTIMATE	48.16	42.99	46.92	35.48	344.67	101.49	126.94	154.11
WikiDocLinks	BINARY	6.35	5.96	6.06	5.13	451.71	34.00	63.00	92.69
($\lambda = 200$)	BIT	6.56	6.13	6.17	5.14	560.53	142.82	171.81	201.51
	BSTL	4.38	4.12	4.16	3.79	553.05	135.34	164.34	194.03
	BY	4.75	4.46	4.60	4.01	451.71	34.00	63.00	92.69
	FS	0.58	0.57	0.56	0.55	793.49	375.77	404.77	434.46
	GALLOP	7.50	7.05	7.25	5.97	451.71	34.00	63.00	92.69
	GDHASH	12.68	12.23	12.47	9.31	758.29	340.58	369.57	399.27
	STL	3.03	2.93	2.93	2.70	451.71	34.00	63.00	92.69
	ULTIMATE	11.20	10.80	11.07	8.47	506.52	88.81	117.81	147.50
Wikipedia	BINARY	102.90	99.74	99.90	91.59	101.49	11.87	21.26	45.78
($\lambda = 500$)	BIT	121.62	115.34	119.28	106.74	153.06	63.44	72.83	97.35
	BSTL	33.88	33.05	33.60	32.65	144.58	54.96	64.35	88.87
	BY	85.20	84.62	82.07	78.22	101.49	11.87	21.26	45.78
	GALLOP	108.99	102.43	104.76	95.90	101.49	11.87	21.26	45.78
	STL	6.90	6.89	6.87	6.81	101.49	11.87	21.26	45.78
	ULTIMATE	301.26	295.66	301.66	250.96	158.64	69.02	78.41	102.93

6 Conclusions

In this paper we have examined the set intersection counting problem, in which we are given a collection of sets of integers, and asked to preprocess the sets so that for any pair we can return the *exact size* of their intersection. This problem appears as a subroutine in many techniques used in natural language processing, in which document cooccurrence counts between pairs of noun phrases or entities are typically used to compute similarity. Using data sets created from real text corpora we show that it is possible to achieve mean query times that are comparable to approximation techniques, such as `MINHASH`.

We have made the following key observations and contributions:

1. Precomputation can result in a large speedup (more than a factor of 50 for the search and hashing-based algorithms) for several state-of-the-art algorithms. Moreover, the matrix of precomputed solutions appears to be highly compressible for the specific scenario of noun phrase document-level cooccurrence.

2. The ranking of the state-of-the-art algorithms (from fastest to slowest) changes significantly when the algorithms are combined with precomputation. Without precomputation, algorithms based on linear merging are fastest, but with precomputation, algorithms based on searching or hashing are faster.

3. We defined a strategy called `ULTIMATE` which combines linear merging, hashing, precomputation and compression with direct access in order to achieve a speedup factor of over 100, while only consuming about twice the memory of baseline approaches.

There are many possibilities for future work:

1. Our framework for comparison is very general, and as such we would like to include additional data sets. Many techniques make use of graph data sets and measure intersection between neighbourhoods (such as one or two-hop neighbours of pairs of nodes), and use this as a distance measure. It would be interesting to include such graphs as they may have different distributions than the presented text-based data sets.

2. As discussed in the experimental section, we experimented with various reordering strategies on the posting lists, as earlier work [35] had found that this can impact the running time significantly. We note that it is also possible to reorder the precomputed matrix of solutions, and such a reordering may increase its compressibility.

3. It would be interesting to do a thorough grid-search for our `ULTIMATE` strategy, to better characterize the trade-off that is available.

4. Finally, since many of the algorithms rely on low-level specialized hardware operations (such as *POPCNT*), it would be interesting to expand the comparison to other hardware architectures.

References

[1] Ricardo A. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In Süleyman Cenk Sahinalp, S. Muthukrishnan, and Ugur Dogrusöz, editors, *Combinatorial Pattern Matching, 15th Annual Symposium, CPM 2004, Istanbul,Turkey, July 5-7, 2004, Proceedings*, volume 3109 of *Lecture Notes in Computer Science*, pages 400–408. Springer, 2004.

[2] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, and Alejandro Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics*, 14, 2009.

[3] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–87, 1976.

[4] Philip Bille, Anna Pagh, and Rasmus Pagh. Fast evaluation of union-intersection expressions. In Takeshi Tokuyama, editor, *Algorithms and Computation, 18th International Symposium, ISAAC 2007, Sendai, Japan, December 17-19, 2007, Proceedings*, volume 4835 of *Lecture Notes in Computer Science*, pages 739–750. Springer, 2007.

[5] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. Dacs: Bringing direct access to variable-length codes. *Inf. Process. Manage.*, 49(1):392–404, 2013.

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

[7] Stefan Büttcher and Charles L. A. Clarke. Index compression is good, especially for random access. In Mário J. Silva, Alberto H. F. Laender, Ricardo A. Baeza-Yates, Deborah L. McGuinness, Bjørn Olstad, Øystein Haug Olsen, and André O. Falcão, editors, *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007*, pages 761–770. ACM, 2007.

[8] Moses Charikar, Kevin C. Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.

[9] Kenneth Ward Church and Patrick Hanks. Word association norms, mutual information, and lexicography. *Comput. Linguist.*, 16(1):22–29, March 1990.

[10] Rudi Cilibrasi and Paul M. B. Vitányi. The google similarity distance. *IEEE Trans. Knowl. Data Eng.*, 19(3):370–383, 2007.

[11] Hagai Cohen and Ely Porat. Fast set intersection and two-patterns matching. *Theor. Comput. Sci.*, 411(40-42):3795–3800, 2010.

[12] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[14] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In David B. Shmoys, editor, *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA.*, pages 743–752. ACM/SIAM, 2000.

[15] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Experiments on adaptive set intersections for text retrieval systems. In Adam L. Buchsbaum and Jack Snoeyink, editors, *Algorithm Engineering and Experimentation, Third International Workshop, ALENEX 2001, Washington, DC, USA, January 5-6, 2001, Revised Papers*, volume 2153 of *Lecture Notes in Computer Science*, pages 91–104. Springer, 2001.

[16] Paul F. Dietz, Kurt Mehlhorn, Rajeev Raman, and Christian Uhrig. Lower bounds for set intersection queries. In Vijaya Ramachandran, editor, *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas.*, pages 194–201. ACM/SIAM, 1993.

[17] Bolin Ding and Arnd Christian König. Fast set intersection in memory. *PVLDB*, 4(4):255–266, 2011.

[18] Benjamin Van Durme and Ashwin Lall. Probabilistic counting with randomized storage. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 1574–1579, 2009.

[19] Benjamin Van Durme and Ashwin Lall. Streaming pointwise mutual information. In Yoshua Bengio, Dale Schuurmans, John D. Lafferty, Christopher K. I. Williams, and Aron Culotta, editors, *Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems 2009. Proceedings of a meeting held 7-10 December 2009, Vancouver, British Columbia, Canada.*, pages 1892–1900. Curran Associates, Inc., 2009.

[20] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

[21] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In Susan Darling Urban and Elisa Bertino, editors, *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, pages 370–379. IEEE Computer Society, 1998.

[22] Amit Goyal, Jagadeesh Jagarlamudi, Hal Daumé, III, and Suresh Venkatasubramanian. Sketching techniques for large scale nlp. In *Proc. NAACL HLT*, WAC-6 '10, pages 17–25, 2010.

[23] Le Quan Ha, E. I. Sicilia-Garcia, Ji Ming, and F. J. Smith. Extension of zipf's law to words and phrases. In *Proceedings of the 19th International Conference on Computational Linguistics*, COLING '02, pages 1–6, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[24] Ioana Hulpus, Conor Hayes, Marcel Karnstedt, and Derek Greene. An eigenvalue-based measure for word-sense disambiguation. In G. Michael Youngblood and Philip M. McCarthy, editors, *Proceedings of the Twenty-Fifth International Florida Artificial Intelligence Research Society Conference, Marco Island, Florida. May 23-25, 2012.* AAAI Press, 2012.

[25] Ioana Hulpus, Conor Hayes, Marcel Karnstedt, and Derek Greene. Unsupervised graph-based topic labelling using dbpedia. In Stefano Leonardi, Alessandro Panconesi, Paolo Ferragina, and Aristides Gionis, editors, *Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, February 4-8, 2013*, pages 465–474. ACM, 2013.

[26] Samuel Huston, J. Shane Culpepper, and W. Bruce Croft. Indexing word sequences for ranked retrieval. *ACM Trans. Inf. Syst.*, 32(1):3, 2014.

[27] Frank K. Hwang and Shen Lin. A simple algorithm for merging two disjoint linearly-ordered sets. *SIAM J. Comput.*, 1(1):31–39, 1972.

[28] Hongrae Lee, Raymond T. Ng, and Kyuseok Shim. Power-law based estimation of set similarity join size. *PVLDB*, 2(1):658–669, 2009.

[29] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1):1–29, 2015.

[30] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. SIMD compression and the intersection of sorted integers. *Softw., Pract. Exper.*, 46(6):723–749, 2016.

[31] Dekang Lin and Xiaoyun Wu. Phrase clustering for discriminative learning. In Keh-Yih Su, Jian Su, and Janyce Wiebe, editors, *ACL 2009, Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing of the AFNLP, 2-7 August 2009, Singapore*, pages 1030–1038. The Association for Computer Linguistics, 2009.

[32] Julian J. McAuley and Jure Leskovec. Hidden factors and hidden topics: understanding rating dimensions with review text. In Qiang Yang, Irwin King, Qing Li, Pearl Pu, and George Karypis, editors, *Seventh ACM Conference on Recommender Systems, RecSys '13, Hong Kong, China, October 12-16, 2013*, pages 165–172. ACM, 2013.

[33] Qiaozhu Mei, Xuehua Shen, and ChengXiang Zhai. Automatic labeling of multinomial topic models. In Pavel Berkhin, Rich Caruana, and Xindong Wu, edi-

tors, *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 490–499. ACM, 2007.

[34] Rasmus Pagh, Morten Stöckel, and David P. Woodruff. Is min-wise hashing optimal for summarizing set intersection? In Richard Hull and Martin Grohe, editors, *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 109–120. ACM, 2014.

[35] Weixiong Rao, Lei Chen, Pan Hui, and Sasu Tarkoma. Bitlist: New full-text index for low space cost and efficient keyword search. *PVLDB*, 6(13):1522–1533, 2013.

[36] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *SIGIR 2002: Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 11-15, 2002, Tampere, Finland*, pages 222–229. ACM, 2002.

[37] Daisuke Takuma and Hiroki Yanagisawa. Faster upper bounding of intersection sizes. In Gareth J. F. Jones, Paraic Sheridan, Diane Kelly, Maarten de Rijke, and Tetsuya Sakai, editors, *The 36th International ACM SIGIR conference on research and development in Information Retrieval, SIGIR '13, Dublin, Ireland - July 28 - August 01, 2013*, pages 703–712. ACM, 2013.

[38] Yufei Tao, Cheng Sheng, Chin-Wan Chung, and Jong-Ryul Lee. Range aggregation with set selection. *IEEE Trans. Knowl. Data Eng.*, 26(5):1240–1252, 2014.

[39] Shirish Tatikonda, Flavio Junqueira, Berkant Barla Cambazoglu, and Vassilis Plachouras. On efficient posting list intersection with multicore processors. In James Allan, Javed A. Aslam, Mark Sanderson, ChengXiang Zhai, and Justin Zobel, editors, *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2009, Boston, MA, USA, July 19-23, 2009*, pages 738–739. ACM, 2009.

[40] Mikkel Thorup. High speed hashing for integers and strings. *CoRR*, abs/1504.06804, 2015.

[41] Peter D. Turney. Mining the web for synonyms: PMI-IR versus LSA on TOEFL. In Luc De Raedt and Peter A. Flach, editors, *Machine Learning: EMCL 2001, 12th European Conference on Machine Learning, Freiburg, Germany, September 5-7, 2001, Proceedings*, volume 2167 of *Lecture Notes in Computer Science*, pages 491–502. Springer, 2001.

[42] Peter D. Turney. Thumbs up or thumbs down? semantic orientation applied to unsupervised classification of reviews. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA.*, pages 417–424. ACL, 2002.

[43] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition.* Morgan Kaufmann, 1999.

[44] Daniel M. Yellin. Representing sets with constant time equality testing. In David S. Johnson, editor, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California.*, pages 64–73. SIAM, 1990.

[45] Wen Zhang, Taketoshi Yoshida, Xijin Tang, and Tu Bao Ho. Improving effectiveness of mutual information for substantival multiword expression extraction. *Expert Syst. Appl.*, 36(8):10919–10930, 2009.