HPTA: High-Performance Text Analytics

Vandierendonck, H., Murphy, K., Arif, M., & Nikolopoulos, D. S. (2017). HPTA: High-Performance Text Analytics. In Proceedings of tge IEEE International Conference on Big Data (pp. 416-423). IEEE . DOI: 10.1109/BigData.2016.7840632

Published in:
Proceedings of tge IEEE International Conference on Big Data

Document Version:
Peer reviewed version

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

HPTA: A Library for High-Performance Text Analytics

Hans Vandierendonck, Karen Murphy, Mahwish Arif, Dimitrios S. Nikolopoulos
School of Electrical Engineering, Electronics and Computer Science
Queen's University Belfast
Email: {h.vandierendonck,k.l.murphy,m.arif,d.nikolopoulos}@qub.ac.uk

May 14, 2016

Abstract

One of the main targets of data analytics is unstructured data, which primarily involves textual data. High-performance processing of textual data is non-trivial. We present the HPTA library for high-performance text analytics. The library helps programmers to map textual data to a dense numeric representation, which can be handled more efficiently. HPTA encapsulates three performance optimizations: (i) efficient memory management for textual data, (ii) parallel computation on associative data structures that map text to values and (iii) optimization of the type of associative data structure depending on the program context. We demonstrate that HPTA outperforms popular frameworks for text analytics such as scikit-learn and Spark.

1 Introduction

Text analytics are an important class of data analytics, differentiated from analytics in general by assigning meaning to *textual data*. Rumor has it that 80% of all big data is unstructured, hence textual in nature. This is however hard to confirm [7]. There is quantitative support for a 31% stake of textual data [7, 27], which is nonetheless a substantial fraction of data analytics.

Analyzing data at high speed is immensely important given that data volumes are consistently growing. The dominant approach to scaling up analytics capabilities consists of using increasing numbers of servers. Single-thread performance, i.e., the time it takes an individual server to process its part of the work, is generally neglected. This approach is not scalable in the long term due to operational costs of the high number of components involved and the diminishing returns that result from scaling up to high degrees of parallelism. In contrast, improving the performance of analytics can reduce operational costs even in the face of growing data volumes.

The data analytics literature generally pays little attention to single-thread performance. There is a good motivation for this: single-thread code is typically written by data analysts and it is not desirable to require high-performance computing expertise from data analysts. In contrast, performance-critical code is encapsulated in libraries and frameworks, although the performance of these is under scrutiny [11, 20–22, 29]. However, to the best of our knowledge, there exist no frameworks, nor good practice, for manipulating textual data *at high speed* for general algorithms. The goal of this work is to fill this gap in the literature and

provide guidelines for achieving *high-performance text analytics*. Moreover, we present HPTA, a library that implements these guidelines in a reusable way.

This paper presents three guidelines towards high-performance text analytics:

Memory management: text analytics will deal with a large number of text fragments. These fragments are often short, e.g., words in a natural language. Traditional memory management, involving independent allocation of each fragment, is not scalable due to the performance overhead of fine-grain dynamic memory management and the resulting fragmentation. Nonetheless, popular systems such as Hadoop [9] and Spark [32] follow this approach. We investigate techniques to circumvent this problem and experimentally characterize their effectiveness.

Parallelism in associative data structures: associative data structures track the computed values for each text fragment. It is well known that the choice of associative data structure, e.g., hash table versus map, affects performance. Data analytics frameworks have settled on lists of key-value pairs as the main associative data structure [9, 40]. Few would argue that this is optimal in single-threaded applications, yet it seems to work well for parallel execution, in particular for data partitioning and reduction. In contrast, frameworks that use more complex data structures are restricted to single-threaded execution [10, 23]. We argue that parallel execution is possible using any associative container and we present methods for partitioning and reduction. Experimental validation shows that the use of appropriate data structures outperforms the list-based representation.

Moving data is faster: We demonstrate that different phases in text analytics applications utilize the data structures in different ways. As such, phases require different data structures, which leads to the counter-intuitive result that moving the data to different data structures during the computation reduces execution time, even though data volumes are large.

The remainder of this paper is organized as follows. Section 2 discusses related work on text analytics and relevant high-performance computing techniques. Section 3 describes the computation of *term frequency-inverse document frequency* (TF-IDF) as a guiding example. Section 4 presents our performance optimization guidelines and their implementation in the HPTA library. Section 5 shows the experimental evaluation of the guidelines. Section 6 concludes the paper.

2 Related Work

Research into high-performance text analytics often involves acceleration using Graphics Processing Units (GPUs). Erra *et al* [4] present a GPU implementation of an approximate streaming version of TF-IDF. The TF-IDF metric is approximated by counting occurrences of a pre-set number of terms only in order to meet the memory limitations of GPUs. They use the C++ Thrust library that simplifies the development of CUDA code.

Zhang *et al* [41, 42] study document clustering on clusters of computers equipped with GPUs. They use the Term Frequency-Inverse Corpus Frequency (TF-ICF) algorithm [26]. TF-ICF approximates the IDF scores using document relevance metrics that are pre-calculated over a reference corpus. They pre-compute TF-ICF scores on the CPU and accelerate a *flock clustering* algorithm on the GPU. They demonstrate a 10x speedup when using 16 high-end NVIDIA GPUs compared to executing on a single desktop.

Szaszy *et al* accelerate document *stream* clustering where they assume that a stream of

documents needs to be continuously clustered [34]. They use sparse matrix-vector multiply (SpMV) techniques to compute the similarity between the TF-IDF of a document and the reference clusters. The SpMV calculation is performed on the GPU. They do not study the issue of text parsing and assume that a document is converted to TF-IDF form prior to entering their system.

Each of the above works investigates acceleration of document clustering and focuses on the numeric clustering algorithm rather than the processing of textual data.

Suffix arrays are a representation of documents that facilitate counting the frequency of terms [18]. A suffix array consists of every suffix of a document, i.e., a sub-string starting at a position in the text and extending to the end of the document. These suffixes are sorted alphabetically. Linear-time traversal or binary search of the suffix array can efficiently retrieve the frequency of individual terms and of n-grams.

Suffix arrays can be constructed in linear time [14]. Combined with a linear-time scan to count term frequencies, suffix arrays promise a linear-time computation of TF-IDF scores. We have experimented with suffix arrays. However, the algorithm we tried [14] is much slower than the solutions we propose in this work. The reason is that, although it is a linear time algorithm, it performs redundant work. The algorithm needs to sort all suffixes, including those starting at arbitrary character positions within terms. The number of suffixes sorted is thus at least an order of magnitude larger than the number of terms. Moreover, the number of terms is typically several orders of magnitude larger than the number of unique terms in a document, leading to further redundant work. As such, these algorithms are efficient only in those cases where all suffixes are required.

Kulla *et al* demonstrated that the DC3 linear-time suffix array construction algorithm [14] can be efficiently parallelized, leading to significant performance gains [17].

Yamamoto and Church propose an algorithm building on suffix arrays that identifies clusters of related terms [39]. Their algorithm does not cluster TF-IDF scores using generic algorithms such as K-means cluster analysis. Instead, they analyze the longest common prefixes of groups of suffixes. They identify clusters of interest as groups of suffixes that have the same term frequency and document frequency.

An important component of this work is concerned with parallel operation on associative data structures. Several works have investigated scalable parallel data structures. The Standard Template Adaptive Parallel Library (STAPL) [25, 36] distributes data structures across a cluster by partitioning the key space. Accesses to data structures are transparently forwarded to the appropriate machine. The Parallel Standard Template Library (PSTL) [13] is a similar, older project. The parallel-STL approach has limited scalability in comparison to this work as it aims to parallelize individual operations on associative data structures. This work, in contrast, is concerned only with parallel iteration.

PDQCollections [38] processes associative data structures in a map-reduce-like model. The authors consider functions on the data such that the data may be split (e.g., by key range), operated on independently and then merged as in a reduction operation. PDQCollections is more akin to the approach taken in this work due to the reduction of associative data sets. An important distinction is that our approach is not specific nor limited to map-reduce programs.

Table 1: Common operations on associative containers.

Operation	Description
insert(c, k, v)	insert value v for key k in collection c
modify(c, k, f, v)	modify collection c to store value v_0 for key k as $v_0 = f(v_0, v)$ or insert v if key k absent
lookup(c, k)	lookup what value is stored for key k in collection c
iterate-seq(c, k, v)	retrieve the next stored key-value pair
iterate-par(c, k, v)	as *iterate-seq*(k, v) but can be used as iterator in a parallel for-loop
merge(c_l, c_r, f, g)	merge collection c_r into c_l by storing the value $f(v_l, v_r)$ if $(k, v_l) \in c_l$ and $(k, v_r) \in c_r$ for a key k, or by inserting $(k, g(v_r))$ for $(k, v_r) \in c_r$.
sort-by-key(c)	sort all entries of collection c by key
sort-by-value(c)	sort all entries of collection c by value

3 Text Analytics: TF-IDF Case Study

To simplify the exposition, we will study term frequency-inverse document frequency (TF-IDF) [28] as a guiding example of text analytics. While the TF-IDF operation is simple enough to understand in detail, it exposes important reoccurring properties of text analytics operations. Before we present the example, we define the operations on associative containers that we will use.

3.1 Operations on Associative Containers

Associative containers present a typical set of operations that relate to their key function: associating a value to a key taking from a sparsely used range of values. As such, the simple operations are *insert*, to insert a value for a key; *lookup*, to lookup the value stored for a key; and *modify*, to modify the value stored for a key. The *modify* operation moreover stores a key-value pair in case the key was absent. Specifics of the function are provided in Table 1.

Associative containers moreover provide means to iterate over all of their contents. The *iterate-seq* operation allows to iterate over all key-value pairs in a sequential (single-threaded) piece of code. Some containers guarantee to access their contents in a specific order. E.g., a tree-based hash map typically stores its contents using a user-specified *less-than* comparator. Its elements are thus iterated in increasing order. In our case, this means terms are iterated through in alphabetic order. Other containers, e.g., a hash table, do not provide this functionality. Regardless of sorting guarantees, we assume that multiple traversal over the same container instance traverse the key-value pairs in the same order.

The *iterate-par* operation allows the same from a parallel loop. Not all containers allow easy parallel access. Typically array-like containers do, while tree-based containers do not.[1]

Text analytics require a few complex operations that manipulate containers as a whole, rather than providing access to individual elements. The *merge* operation merges the contents of one container into the other. This is useful to combine the information from multiple containers. E.g., we use the *merge* operation to compute the document frequency from per-document term frequency containers. Finally, the *sort* operations sort the contents of the

[1] We follow C++ terminology and assume that containers with a *random access iterator* provide the *iterate-seq* and the *iterate-par* operation, while others only support the *iterate-seq* operation [3].

container. Sorting can be done either by key, which is the natural way to store the contents for some container types, or sorting can be done by value. Sorting is not supported by all containers.

3.2 TF-IDF

TF-IDF assigns a weight to each term-document combination. The weights reflect the importance of the term within the document and across the set of documents.

Figure 1 shows a pseudo-code for TF-IDF. This code uses a number of associative containers that store information on each encountered term. Firstly, the code uses an associative container per input file to store the term frequency within that document. I.e., the container associates every term (key) to its frequency of occurrence (value). Secondly, a single associative container is used to calculate the document frequency across the collection. This container stores two integer values for each term encountered in each of the documents: the number of documents where the term occurs (document frequency) and a unique sequence number that is determined only when all files have been read. The latter number is important for sorting the output data alphabetically.

The algorithm has three distinct phases. In the first phase (term count phase), all documents are read and the per-document term frequency is determined. Moreover, all terms from all documents are added to the document frequency container and the document frequency is updated. The containers are mostly updated during the term count phase. The access pattern consists thus of random accesses.

The second phase of the algorithm assigns a unique ID to each term. This is helpful to build the TF-IDF matrix, i.e., to index it by numeric ID rather than by string. Assigning IDs is however also critical in order to produce an alphabetically sorted output.

The third phase computes the TF-IDF scores and stores them in a matrix. Although the pseudo-code depicts a dense matrix, a sparse matrix is used as non-stop-words typically occur in only a fraction of the documents. The matrix is built up by rows, where rows can be easily constructed in parallel. Each row corresponds to a document and is constructed by iterating over all elements of the corresponding per-document term frequency container. Each term in this container is looked up in the term frequency container to obtain the corresponding document frequency.

3.3 Discussion

3.3.1 Memory Management

TF-IDF constructs a *bag-of-words* model for a set of documents. It extracts individual words from the documents and retains one copy of each unique word. As such, the final set of words is significantly smaller than the sum of document sizes.

A typical implementation in Hadoop or Spark would allocate memory for every word separately as it is obtained from the tokenizer. These words will then be added to the associative container that counts occurrences of the words. When the text fragment is already present, the new copy will be discarded. As such, some words are discarded quickly, others are retained for a longer period of time.

The problem arises as natural language words are typically short, there are many words in large documents and the words have varying lengths and lifetimes. This poses issues for the

```
1  procedure TFIDF(documents[0..n−1]) {
2      // term frequency per document
3       associative_container ( string  −> int) term_freq[n];
4      // document freq. and ID
5       associative_container ( string  −> (int,int))  doc_freq ;
6       parallel_for ( i  :  0.. n−1) {
7           // Calculate term frequency in i−th document
8            parallel_for (term : documents[i])
9              modify(term_freq [ i ], term,+,1)
10          // Update document frequencies for term in i−th document
11          // Increment counter for each term ignoring term frequency
12          // Value of ID is irrelevant at this time
13          merge(doc_freq, term_freq [ i ],
14                  f=(k,( dfl , idl ), tfr )−>(k,(dfl+1, idl )),
15                  g=(k, tfr )−>(k,(1,0)))
16     }
17     // Assign unique IDs to each term. The terms can be optionally
18     // sorted alphabetically . Sorting here affects the order of
19     // terms in the TF−IDF matrix and output.
20     // Store IDs in second element of value pair in doc_freq .
21     sort −by−key(doc_freq)
22     ID = 0;
23      parallel_for (term : doc_freq ) {
24          modify(doc_freq , term, f=((tf , old_ID ), ID)−>(tf,ID))
25          ID += 1
26     }
27     // Construct TF−IDF (sparse) matrix
28      parallel_for ( i  :  0.. n−1) {
29          for((term, tf ) : term_freq [ i ]) {
30              // Calculate TF−IDF score for term in i−th document
31              (df , id ) := lookup(doc_freq , term)
32               tfidf [ i , id ] := tf ∗ log((df+1)/(n+1))
33          }
34     }
35     return  tfidf
36 }
```

Figure 1: Code structure for term frequency–inverse document frequency calculation for a collection of documents. The operations on associative containers are defined in Table 1.

efficiency of memory management, such as time spent in (de-)allocation or garbage collection and fragmentation of memory [15].

3.3.2 Parallelism

There is abundant parallelism in TF-IDF. In fact, *all* of the loops in Figure 1 can be executed in parallel. Some loops are trivial to parallelize, while others require more work. For instance, the loop at Line 8 can be parallelized by dividing the document in large chunks, split at a word boundary [24]. Distinct associative containers are computed for each chunk. These are reduced in pairs using a tree reduction at the end of the loop. Moreover, the loop at Line 23 can be parallelized using a prefix sum [1].

Loops may be more or less difficult to parallelize depending on the data structures involved.

E.g., parallel iteration over an array-based list can be trivially achieved by splitting the index range. Iterating over the elements of a hash table in parallel is also possible but needs to tie in with the bucket structure of the hash table. On the other hand, parallel iteration over self-organizing data structures such as splay trees [31] is incompatible with the ongoing re-balancing.

3.3.3 Containers

The TF-IDF operator code is heavily dependent on accesses to the containers in order to accumulate and retrieve term frequencies and document frequencies. Our work builds on the observation that execution time is determined by the characteristics of the container. We strive to optimize the performance of text analytics in general and TF-IDF in particular by selecting the most appropriate type of container for each phase of the algorithm.

4 Optimization of Text Analytics

We have identified optimization opportunities that are applicable to text analytics operations in general, and to TF/IDF specifically. These optimizations are implemented in the HPTA library. We will experimentally demonstrate their impact in Section 5.

4.1 Memory Management

Text analytics operate on a large collection of text fragments. A common goal is to map these text fragments into a numeric multi-dimensional space [28], but until that mapping is achieved, text analytics operations process individual text fragments. The text fragments can be created and represented in multiple ways:

Text fragments are individually allocated as they are read in or discovered. Memory allocators typically round allocated memory sizes up to frequently occurring sizes. This will incur significant internal fragmentation as text fragments have highly varying lengths. Alternatively, systems using garbage collection will incur significant garbage collection overheads when all text fragments are separately allocated. The garbage collector must analyze these objects upon each collection pass, adding to the overhead of garbage collection [33]. However, it can be expected that large groups of text fragments have equal life-times in text analytics applications.

The input files are retained integrally. A fast solution results when reading in input files integrally into working memory [24], e.g., using `mmap` on UNIX-based systems. This avoids separate memory allocations for each fragment in the input. However, it will result in large memory overhead and bad memory locality. In particular, when terms repeat many times in the same document, each repetition of the word will be held in memory while a *bag-of-words* model requires that only one copy of each word is stored. This is the case, e.g., in the TF-IDF example. More importantly, retaining full input files requires that sufficient main memory is available.

Region-based memory allocators aim to maximize performance by eradicating internal fragmentation and by efficiently de-allocating a large number of items in bulk [6, 12, 33]. Region-based memory allocation is effective when individually allocated items go out of scope

at the same time, implying that their memory can be reclaimed at the same time. Region-based memory management is more sophisticated than retaining all input files in memory, but results in similar performance benefits.

Region-based memory management is generally provided using application-specific code [6, 12]. Language support has been proposed [8,30] but is not widely available. As such, we have selected a library implementation.

4.1.1 HPTA Implementation

HPTA implements a *word bank*, which is a data structure that implements region-based memory allocation of strings. The word bank consists of a list of large chunks of memory that are allocated using the system-supplied memory allocator. Words that are added to the word bank are allocated at the end of the last chunk using a simple bump-pointer allocation technique. When all memory in the last chunk has been used, or the next string is too long to fit in the chunk, a new chunk is appended to the list. The chunk size can be tuned by the programmer.

The word bank has the advantage that it limits the run-time overhead and fragmentation of the system memory allocator. Moreover, all words contained in it can be de-allocated quickly. In general, using larger chunk sizes results in less system overhead. A downside of the word bank is that individual strings cannot be removed or de-allocated. This is a known limitation of region-based memory management [6, 8, 12]. We believe that this limitation is however not important for text analytics as remove operations are rare in this area. They are also not included in Table 1. If, however, a significant number of strings needs to be removed, it is viable to copy over the remaining strings to a new word bank and discard the old one, much like a copying garbage collector works.

HPTA furthermore couples each associative data structure with a word bank. As such, the associative data structure and its word bank are created and destroyed together. The main advantage of this approach is that it is safe to store pointers to strings in the associative data structure where the pointers point into the word bank. The validity of the pointers follows trivially from this setup.

In some applications it is advantageous to share a text corpus between multiple associative data structures. HPTA supports this by allowing word banks to share memory chunks. Memory reclamation is controlled by adding reference counters to the memory chunks. As such, the chunks are de-allocated when all word banks that contain it are destroyed.

4.2 Reference Associative Containers

A myriad associative containers have been proposed in the literature, each making specific trade-offs in the time complexity of various operations, in average-case vs. worst-case time complexity, in memory efficiency, in raw performance, etc. The goal of this work is not to identify the best possible container for text analytics or for TF-IDF. Rather, we aim (i) to demonstrate that text analytics are extremely sensitive to the properties of the containers,

[2]Time complexity is $\mathcal{O}(n)$ if key is new due to moving elements in the array.

[3]Assumes usage of the C++'11 insertion hint indicating that the element is inserted in the immediate neighborhood of an iterator. The iterator is assumed to be the position where the previous element was inserted.

Table 2: Time complexity of operations on associative containers assuming the container holds n elements.

Operation	**Time Complexity**			
	List	**Sorted List**	**Hash Table**	**Map**
insert(k, v)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
modify(k, f, v)	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$[2]	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
lookup(k)	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
iterate-seq(k, v)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
iterate-par(k, v)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	n/a	n/a
merge(c_l, c_r, f, g)	$\mathcal{O}(n_l n_r)$	$\mathcal{O}(n_l + n_r)$	$\mathcal{O}(n_r)$	$\mathcal{O}(n_l + n_r)$[3]
sort-by-key(c)	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	n/a	$\mathcal{O}(1)$
sort-by-value(c)	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	n/a	n/a

(ii) identify the opportunity for moving data from one container type to another during an algorithm and (iii) to set out guidelines how to select container types depending on the algorithm.

In the following we consider four different classes of associative containers. These correspond to the basic classes: lists of key-value pairs, sorted lists of key-value pairs, hash tables and hash maps. These are different enough to warrant their study. Other data structures would show similar properties to one of these four classes.

Table 2 shows the average-case time complexity of the common operations performed on associative containers for these 4 data structures. In the case of lists of key-value pairs, we assume that the data is stored in an array-based data structure for reasons of efficiency. For sorted lists of key-value pairs we assume that value lookup uses a binary search algorithm [16]. The time complexity of the hash table is based on the unordered map described in the C++ standard [3], while the map is based on the C++ map, which always stores its elements in sorted order.

Time complexity alone does not determine performance. Containers with worse time complexity may outperform containers with better time complexity in specific circumstances, e.g., when the container holds few elements. Data analytics, however, are anticipated to execute over large data sets. As such, time complexity is a good first-order approximation of performance. Our experiments confirm that performance can be explained by the qualitative properties of the containers. Performance measurements were required only to break ties between containers with the same time complexity.

Table 2 shows that a hash table provides best time complexity on a range of operations. However, it is not possible to sort the contents of a hash table. In order to do that, it is necessary to move the data to a different container, either a list of key-value pairs or a map. However, once the data has been moved over, all operations have higher time complexity. It is now more expensive to access the data. Hence, a careful trade-off is necessary to decide on conversions.

For completeness, we show the assumed time complexity in Table 3. It is also possible to merge containers of different types. Table 4 shows the time complexity of such merge operations.

[4] Assumes usage again of the C++'11 insertion hint. If absent, time complexity is $\mathcal{O}(n \log n)$.

Table 3: Conversion cost of associative containers holding n elements.

Source	**Target Container**			
	List	**Sorted List**	**Hash Table**	**Map**
List	–	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
Sorted List	$\mathcal{O}(1)$	–	$\mathcal{O}(n)$	$\mathcal{O}(n)^4$
Hash Table	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	–	$\mathcal{O}(n \log n)$
Map	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	–

Table 4: Cost of merging associative containers of different types.

Right (m)	**Left Argument** (n)			
	List	**Sorted List**	**Hash Table**	**Map**
List	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$	$\mathcal{O}(m)$	$\mathcal{O}(m \log n)$
Sorted List	$\mathcal{O}(mn)$	$\mathcal{O}(m+n)$	$\mathcal{O}(m)$	$\mathcal{O}(m)$
Hash Table	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$	$\mathcal{O}(m)$	$\mathcal{O}(m \log n)$
Map	$\mathcal{O}(mn)$	$\mathcal{O}(m+n)$	$\mathcal{O}(m)$	$\mathcal{O}(m)$

4.3 Container Selection

As indicated above, containers must be selected with care, but when done right, there is opportunity for switching containers. In this Section we outline our methodology to select containers. Referring back to the TF-IDF algorithm (Figure 1), we observe that each container is used in different ways throughout the algorithm. The per-document term catalogs are used in line 15 only with the *modify* operation. At lines 9 and 29, the catalogs are traversed sequentially, either in a *merge* operation or using *iterate-seq* during the construction of the TF-IDF matrix. The *modify* operation is clearly most efficient on a hash table (Table 2). Iteration over all elements has $\mathcal{O}(1)$ time complexity for lists and the hash table. Detailed measurement has shown however that iteration through an array-based list is more efficient than through a hash table. As such, we consider that there is opportunity to change the container type for the term catalogs prior to line 15.

Similarly, we analyze the usage of the document frequency container. This container is updated using *merge* at line 15. The merge operation is most efficient when the left-hand-side argument (`doc_freq`) is a hash table (Table 4). In fact, the hash table is the only data structure where the time complexity of *merge* is independent of the size of `doc_freq`. At line 21, however, the document frequency container must be sorted by key, which is impossible with a hash table. A change in container is thus necessary due to the functionality. At line 23, the document frequency container is traversed, preferably in parallel. This is most efficient with a list-based data structure. The subsequent modification is, however, $O(1)$ in all cases as *modify* can be performed through a pointer into the container. Finally, at line 31, a *lookup* of the document frequency is performed, which is again more efficient with a hash table. We have thus identified four code regions accessing the document frequency container. Each code region has a distinct preference for the container type, which can be distinct from that of the preceding code region.

Note that data structure conversions are a non-obvious choice when working with potentially large data sets. In fact, the leading data analytics platforms have designed their parallel

execution exclusively around lists: Hadoop [9] operates exclusively on key-value lists while Spark [32] is organized around resilient distributed data sets (RDDs), which, like our key-value lists, are essentially arrays. Other data structures are second-class citizens: one cannot parallelize operations across their contents and they do not allow data flow optimizations as RDDs do.

4.4 Parallelization

Parallelism occurs naturally in data analytics due to the possibility to traverse data sets in parallel. We claim that the data structure storing the data set is largely irrelevant to traverse it in parallel. While it is clear that an array-based list can be traversed in parallel, so too can any iterable collection. In the worst case, parallel traversal may require additional computation in order to get each parallel thread started. Concretely, for data structures providing a C++ *random access iterator*, such as arrays, we divide the iteration range among the threads. Each thread can jump directly to the appropriate position due to the random access nature of the iterator. For data structures that provide a C++ *input iterator*, we can divide the iteration range similarly on the basis of the number of elements to iterate over. However, finding the appropriate starting point requires repeated increments of the iterator to traverse from the beginning of the collection to the desired point. This can be done, e.g., using `std::advance()` in C++, which is a linear-time operation for input iterators.

Apart from traversing data sets in parallel, we also need to construct associative data structures in parallel. One approach is to use concurrent or parallel data structures where multiple threads can insert or modify elements [13,36]. This approach potentially has limited scalability due to the need to synchronize threads when accessing the shared data structure. The approach chosen in this work is to construct private data structures within each thread and to *merge* these data structures in pairs as threads complete. We demonstrate that this approach results in a high degree of scalability.

5 Experimental Evaluation

We analyze the performance of the proposed optimizations to TF-IDF experimentally on a quad-socket 2.6GHz Intel Xeon E7-4860 v2, totaling 48 threads. The operating system is CentOS 6.5 with the Intel C compiler version 14.0.1. We have implemented HPTA in C++ and parallelized key operations using Cilk [5], using Intel Cilkplus. We implemented TF-IDF using the associative data structures and memory management techniques of HPTA. Reported results are averaged over 10 executions.

We evaluate our optimizations using 4 data sets (Table 5). The data sets are collected from the public domain and have varying sizes. The "artificial" data set has the unique feature that it spends all it time computing term frequencies and negligible time computing document frequencies and TF-IDF scores. This will be helpful in performance analysis.

The evaluation below focuses on the TF-IDF algorithm for words. We have also evaluated the effectiveness of the optimizations when calculating TF-IDF for 3-grams. The results are qualitatively the same. As such we present results only for single-word terms (1-grams).

Table 5: Characterization of input data sets.

Data set	Description	Size	Files	Unique words
Various	"Classic3" and "Classic4" data sets (CISI, CRAN, MED and ACM) and Reuters press releases (Reuters-21578)	62.8 MB	23 K	192 K
NSF Abstracts	NSF research award abstracts 1990–2003 [37]	311 MB	101 K	268 K
Gutenberg	A selection of e-books from Project Gutenberg, covering multiple languages	20.00 GB	52,361	259 M
Artificial	Phoenix++ [35] word count data set. 4th and 5th file repeat 3rd file 4 times, respectively 8 times	1.33 GB	5	144 K

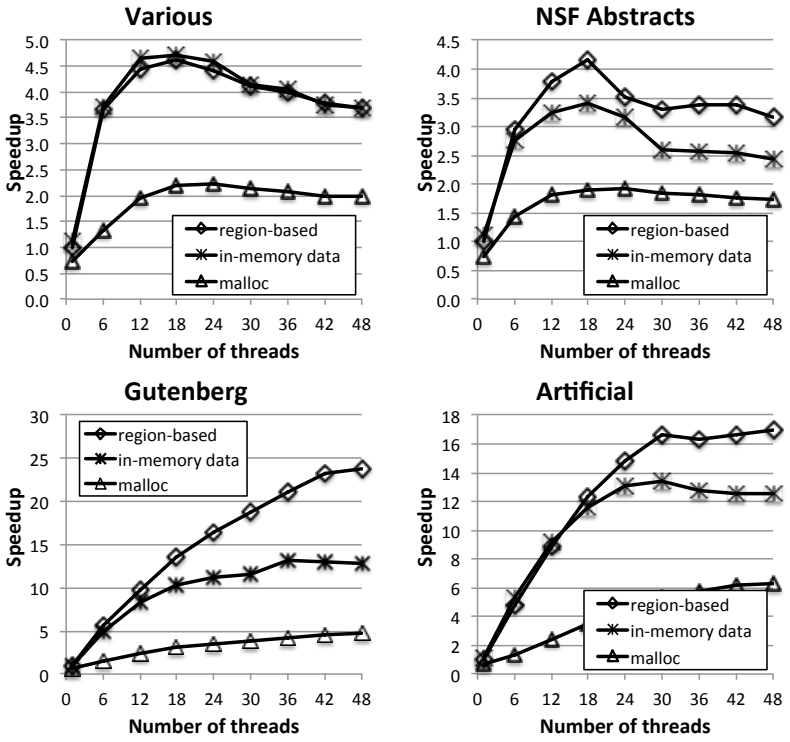

Figure 2: Parallel speedup dependent on the memory management policy. Speedups are normalized against region-based memory management.

5.1 Memory Management

The memory management policy has an important impact on the performance of text analytics. Figure 2 shows parallel speedup using the system memory allocator, region-based memory management and retaining all input files in-memory. All associative data structures

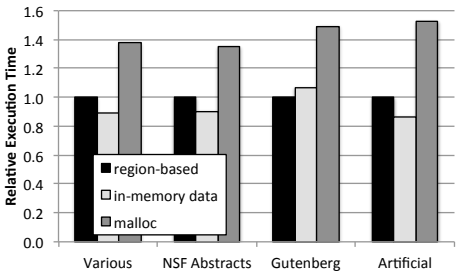

Figure 3: Relative execution time for three memory management policies. Execution times are normalized against region-based memory management. Lower is better.

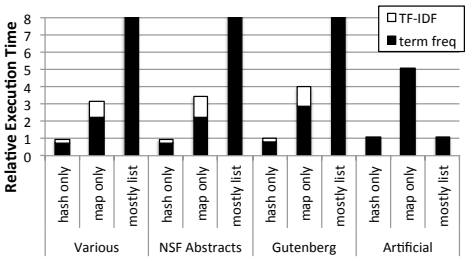

Figure 4: Execution time when storing the term frequency data in a hash table, a key-value list or a map, normalized to the hash table case.

are hash tables in this experiment. Note that parallel speedups range from 4× to 24× and correlate strongly to the data set size.

The system allocator has the lowest performance across the board. It is known that memory-locality-aware and NUMA-aware memory allocators provide higher performance [19], so the performance of per-word memory allocation could be improved on. However, this is not the only issue. Analyzing the single-thread execution time (Figure 3) demonstrates that per-word memory allocation also incurs overhead due to extra work performed.

Keeping all input files in-memory avoids memory allocation as each word can point directly to the input file buffer. This technique is used in the Phoenix work [24, 35]. It is generally faster than the region-based memory allocator for the smaller input files when executing sequentially (Figure 3). For the largest input, it is however slower than the region-based allocator. Moreover, this technique scales not as good with an increasing thread count. This is due to an increased memory footprint, which results in worse locality than the region-based allocator. We consider only the region-based allocator in the remainder of this paper.

5.2 Exploration of Container Types

We analyze performance assuming only one data structure is used throughout the computation. Figure 4 shows the execution time normalized to using a hash table. Results shown correspond to single-threaded execution. The parallel execution supports the same conclusions. We omit the execution times for the sorting stage as this is marginal or not applicable in the case of the hash table.

We conclude that using key-value lists throughout the computation provides really poor performance, up to 20x slower for the NSF Abstracts data set. Note that we used a hash table for computing term frequencies. Otherwise performance would be significantly worse. This is interesting to note as the key-value list abstraction is fundamental to the operation of Hadoop [9] and lies at the heart of Spark's RDDs [32].

The main performance bottleneck in our list-based implementation is the *merge* operation, which has time complexity $\mathcal{O}(m+n)$ to merge collections of n and m elements. Note that *merge* is called once per document and that the size of the target container is continuously growing throughout execution. Assume for the sake of argument that d documents contain m unique words each, then the time complexity of *merge* is $\mathcal{O}(d^2m)$. A Hadoop-like sorting

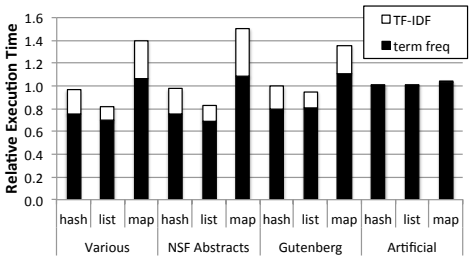

Figure 5: Execution time when retaining the term frequency data in a hash table, or when converting it to a key-value list or a map, normalized to the hash table case. Document frequencies are stored in a hash table.

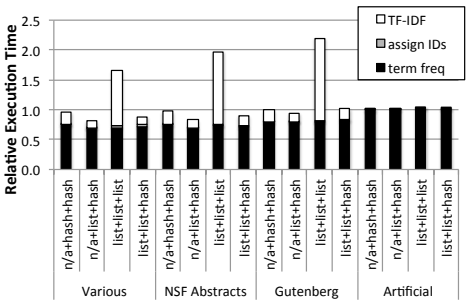

Figure 6: Execution times. Format: sort+iterate+lookup, where sort is the container used to sort words, iterate is the container type iterated during term catalog and lookup is the container type used for document frequency lookup. The remaining operations are performed on hash tables.

solution could perform better with a time complexity of $\mathcal{O}(dm \log dm)$, assuming a list of dm words is first constructed by concatenation and subsequently sorted.

5.3 Unsorted Output

We will first consider the case where the corpus need not be sorted alphabetically. In this case, the sorting step can be omitted and document frequencies can be stored in a hash table throughout the algorithm. We have however observed that execution time can be reduced by converting the term frequency container to a sorted list. Term frequencies are stored in a hash table during construction (Lines 8– 9, Figure 1) and converted to a list prior to Line 13. Figure 5 shows performance when using a hash table, a key-value list or a map for the *merge* and *iterate-seq* operations. Converting the data to key-value list is worthwhile as it is much faster to iterate through a list vs. a hash table. Overall execution time is reduced by up to 19% for the "Various" data set. The "Artificial" data set is slowed down marginally ($< 1\%$) as the conversion takes time and does not lead to significant gains due to the low number of documents.

5.4 Sorted Output

If the output should be alphabetically sorted, it is necessary to assign numeric IDs to terms in alphabetically increasing order. This requires a conversion of document frequencies to a sorted container which, in practical terms, implies a sorted key-value list (results with a map are invariable worse). The TF-IDF phase performs lookups on the document frequencies. These can be performed either using binary search on the list, or on a hash table provided the data is converted back to a hash table. Figure 6 shows these options. The first bar corresponds to using only hash tables. The second bar corresponds to converting term frequencies to a list, the best case for unsorted output. The third bar shows execution time performing lookups using binary search on a sorted key-value list. This is unacceptably slow. The fourth bar

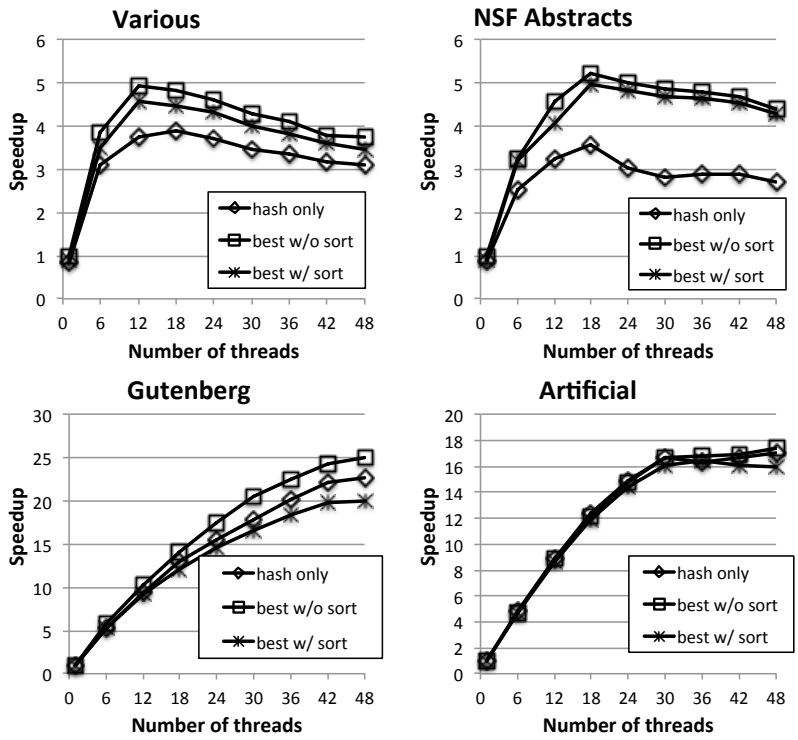

Figure 7: Parallel scalability of TF-IDF normalized against using a hash table for lookup-intensive code regions and a list for iteration-intensive code regions.

shows that converting the document frequencies back to a hash table for fast lookup results in performance competitive with that of the unsorted case, and often out-performs the solution with only hash tables. Yet, the output is alphabetically sorted.

5.5 Parallel Scalability

Using lists rather than hash tables has additional advantages for parallel execution as it is easier and more efficient to parallelize accesses to (array-based) lists. The best version without sorted output achieves higher scalability than the hash table-only version (Figure 7). This furthermore depends on the data set: data sets with few files (Gutenberg and artificial) observe less benefit from converting the term frequencies to lists.

The best algorithm for sorted output can achieve better scalability than the hash table-only version when the number of files is large. It performs poorly on the Gutenberg data set as the number of unique words is very large, which implies more time is spent sorting the corpus.

5.6 Comparison Against Other Systems

We compare the execution time of HPTA with other systems, namely Phoenix++ [35], SciKit-Learn [23] and Spark [32].

Phoenix++ [35] is a shared memory runtime system for map-reduce workloads. We have

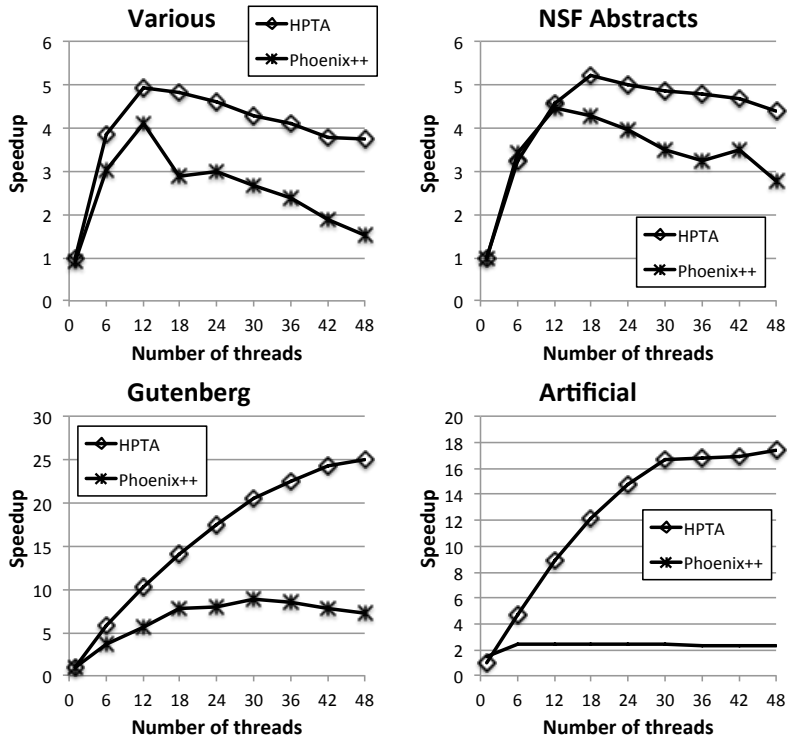

Figure 8: Parallel scalability of TF-IDF comparing the optimized solution against a map/reduce solution using Phoenix++.

Table 6: TF/IDF execution time (seconds) with HPTA, SciKit-Learn and Spark MLlib. T_1 shows single-thread execution time; T_{48} is execution time for 48 threads.

	HPTA	**SKLearn**	**Spark**		
Data set	T_1	T_1	T_1	T_{48}	T_1/T_{48}
Various	1.8	13.4	281.2	207.3	1.4
NSF Abstracts	7.5	44.5	1154.9	888.3	1.3
Gutenberg	385.7	4448.0	1960.5	1211.9	1.6
Artificial	16.2	267.6	GC overhead limit reached		

implemented TF-IDF in Phoenix++ with two map/reduce rounds: one for the word count and one to merge dictionaries and calculate TF/IDF scores. Each document is parsed by one map task, so there is no parallel processing of large files. On the other hand, the map task uses a hash table to store the per-document dictionary. Execution times of HPTA and Phoenix++ are shown in Figure 8. The Phoenix++ code has limited scalability compared to HPTA. This is in part due to explicit serialization and traversal of key-value pairs. Performance on the "Artificial" workload is limited as the word count phase handles each document in a sequential manner.

We also compare against SciKit-Learn, which has only a single-threaded implementation [2]. SciKit-Learn builds on the high-performance NumPy library. Nonetheless, HPTA

is one order of magnitude faster than SciKit-Learn (Table 6). This is because SciKit-Learn is prone to the limitations addressed by HPTA. Moreover, we observe that performance of SciKit-Learn breaks down for the largest, 20 GB, data set.

We furthermore compare to Spark MLlib. Table 6, columns 4–6, show that Spark, executing on the 48-core node, has limited scalability. More importantly, the single-threaded execution time of Spark is extremely high. Even though Spark has been built for scale-out scenarios, it remains important to pay attention to single-thread performance [20].

Results for the "Artificial" benchmark are missing as Spark crashed with an exception due to spending too much time in garbage collection.

6 Conclusion

Text analytics are an important type of data analytics. We address the unexplored issue of manipulating text fragments *at high speed*, which is orthogonal to achieving speed-up by scaling-out analytics processing. The goal of this work is to formulate guidelines for optimizing text analytics and to demonstrate that they can be implemented in a reusable library. We have identified three performance optimizations: (i) region-based memory management, (ii) selection of associative data structures and (iii) transferring between associative data structures throughout the computation. We note that these optimizations are not implemented in leading data analytics platforms such as Hadoop and Spark. Our experimental evaluation however shows significant performance improvements, up to 5× for region-based memory management, up to 20× for data structure optimization and up to 19% for changing data structures during the computation.

Acknowledgement

This work is supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under the ASAP project, grant agreement no. 619706, and by the United Kingdom EPSRC under grant agreement EP/L027402/1.

References

[1] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[2] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *Proceedings of the ECML/PKDD Workshop: Languages for Data Mining and Machine Learning*, page 15, Sept. 2013.

[3] International standard ISO/IEC 14882:2014(E) programming language C++, 2014.

[4] U. Erra, S. Senatore, F. Minnella, and G. Caggianese. Approximate TFIDF based on topic extraction from massive message stream using the GPU. *Information Sciences*, 292:143 – 161, 2015.

[5] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the 1998 ACM SIGPLAN conference on Programming language design and implementation*, pages 212–223, 1998.

[6] D. Gay and A. Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 70–80, New York, NY, USA, 2001. ACM.

[7] S. Grimes. Unstructured data and the 80 percent rule. `http://breakthroughanalysis.com/2008/08/01/unstructured-data-and-the-80-percent-rule/`, Aug. 2008.

[8] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM.

[9] Apache hadoop. `http://hadoop.apache.org`, 2016.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[11] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of Pregel-like graph processing systems. *Proc. VLDB Endow.*, 7(12):1047–1058, Aug. 2014.

[12] H. Inoue, H. Komatsu, and T. Nakatani. A study of memory management for web-based applications on multicore processors. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 386–396, New York, NY, USA, 2009. ACM.

[13] E. Johnson and D. Gannon. HPC++: Experiments with the parallel standard template library. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 124–131, New York, NY, USA, 1997. ACM.

[14] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, Nov. 2006.

[15] D. E. Knuth. *The Art of Computer Programming Volume 1: Fundamental Algorithms.* Addison Wesley, 1997.

[16] D. E. Knuth. *The Art of Computer Programming Volume 3: Sorting and Searching.* Addison Wesley, 1998.

[17] F. Kulla and P. Sanders. Scalable parallel suffix array construction. *Parallel Comput.*, 33(9):605–612, Sept. 2007.

[18] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[19] Y. Mao, R. Morris, and F. Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, MIT Computer Science and Artificial Intelligence Laboratory, 2010.

[20] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[21] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proc. of the 12th USENIX Conf. on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, 2015.

[22] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proc. of the 2009 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '09, pages 165–178, 2009.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, Nov. 2011.

[24] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[25] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard templates adaptive parallel library (STAPL). In *LCR '98: Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 402–409, 1998.

[26] J. W. Reed, Y. Jiao, T. E. Potok, B. A. Klump, M. T. Elmore, and A. R. Hurson. TF-ICF: A new term weighting scheme for clustering dynamic data streams. In *5th International Conference on Machine Learning and Applications (ICMLA'06)*, pages 258–263, Dec 2006.

[27] P. Russom. BI search and text analytics. new additions to the BI technology stack. `http://download.101com.com/pub/tdwi/Files/TDWI_RRQ207_lo.pdf`, 2007.

[28] G. Salton and M. J. McGill, editors. *Introduction to Modern Information Retrieval*. Mcgraw-Hill, 1983.

[29] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proc. of the 2014 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '14, pages 979–990, 2014.

[30] D. Shabalin and M. Odersky. Region-based off-heap memory for Scala. Technical report, École Polytechnique Fédérale de Lausanne, Feb. 2015. Available as `https://infoscience.epfl.ch/record/213469/files/Regions\%20report.pdf`.

[31] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[32] Apache spark. `http://spark.apache.org`, 2016.

[33] C. Stancu, C. Wimmer, S. Brunthaler, P. Larsen, and M. Franz. Safe and efficient hybrid memory management for Java. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 81–92, New York, NY, USA, 2015. ACM.

[34] M. J. Szaszy and H. Samet. Document stream clustering using GPUs. Available at `http://wwwold.cs.umd.edu/Grad/scholarlypapers/papers/Szaszy.pdf`, 2013.

[35] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.

[36] G. Tanase, C. Raman, M. Bianco, N. M. Amato, and L. Rauchwerger. Languages and compilers for parallel computing. chapter Associative Parallel Containers in STAPL, pages 156–171. Springer-Verlag, Berlin, Heidelberg, 2008.

[37] NSF research awards abstracts 1990–2003. `archive.ics.uci.edu/ml/machine-learning-databases/nsfabs-mld/nsfawards.html`, Nov. 2003. Consulted: February 2016.

[38] M. Varshney and V. Goudar. PDQCollections: A data-parallel programming model and library for associative containers. Technical Report 130004, Computer Science Department, Universtiy of California, Los Angeles, Apr. 2013.

[39] M. Yamamoto and K. W. Church. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Comput. Linguist.*, 27(1):1–30, Mar. 2001.

[40] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[41] Y. Zhang, F. Mueller, X. Cui, and T. Potok. Large-scale multi-dimensional document clustering on GPU clusters. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10, April 2010.

[42] Y. Zhang, F. Mueller, X. Cui, and T. Potok. Data-intensive document clustering on graphics processing unit (GPU) clusters. *J. Parallel Distrib. Comput.*, 71(2):211–224, Feb. 2011.