Finding and Exploring Memes in Social Media

Hohyon Ryu, Matthew Lease, and Nicholas Woodward
School of Information
University of Texas at Austin
hohyon@utexas.edu, ml@ischool.utexas.edu, nwoodward@mail.utexas.edu

ABSTRACT

Critical literacy challenges us to question how what we read has been shaped by external context, especially when information comes from less established sources. While cross-checking multiple sources provides a foundation for critical literacy, trying to keep pace the constant deluge of new online information is a daunting proposition, especially for casual readers. To help address this challenge, we propose a new form of technological assistance which automatically discovers and displays underlying *memes*: ideas embodied by similar phrases which are found in multiple sources. Once detected, these underlying memes are revealed to users via generated hypertext, allowing memes to be explored in context. Given the massive volume of online information today, we propose a highly-scalable system architecture based on MapReduce, extending work by Kolak and Schilit [11]. To validate our approach, we report on using our system to process and browse a 1.5 TB collection of crawled social media. Our contributions include a novel technological approach to support critical literacy and a highly-scalable system architecture for meme discovery optimized for Hadoop [25]. Our source code and *Meme Browser* are both available online.

Categories and Subject Descriptors

H.5.4 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia—*Architectures*; H.4.3 [**Information Systems Applications**]: Communications Applications—*Information browsers*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Miscellaneous*

General Terms

Algorithm, Design, Experimentation, Measurement

Keywords

automatic hypertext, critical literacy, memes, MapReduce

1. INTRODUCTION

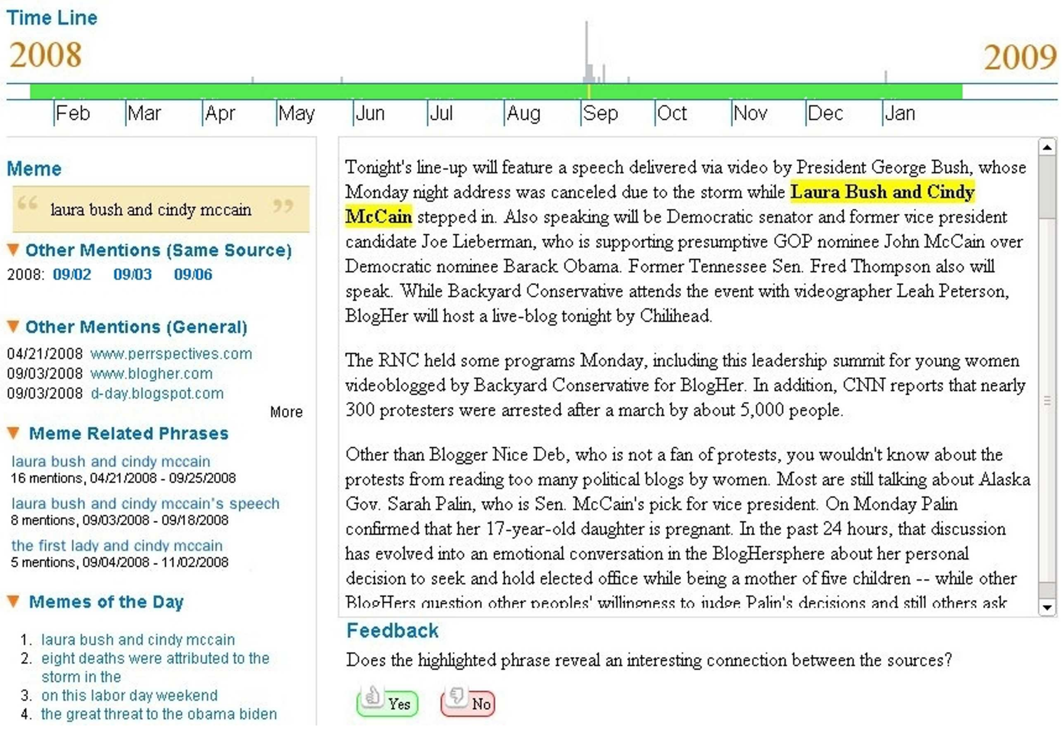
Web 2.0 technologies have broken down many traditional barriers to authorship, enabling greater democratization of information exchange via online social media, where hypertext can blur distinctions between readers and writers [15]. However, the massive growth of information produced by less established sources has created significant new *critical literacy*[1] challenges for helping people effectively interpret and assess online information [24, 5]. Our vision is to complement traditional forms of critical literacy education with additional technological support in the form of smarter browsing technology. Such technology has additional promise for providing new paths for discovering and exploring collections [19], as well as integrating information expressed across multiple sources. Instead of understanding online narrative through only a single source, we can instead explore how broader community discourse has shaped its development [2] or facilitates greater social engagement [9].

Our analysis centers on fine-grained discovery and display of *memes*: ideas represented by one or more similar phrases which occur across multiple collection documents. While our work is inspired by Leskovec et al.'s system for meme detection and visualizion [12], their notion of memes was restricted to explicit quotations only, ignoring the vast majority of text. This assumption is limiting with social media, which often eschews use of quotations for reasons ranging from innocuous social norms to more insidious "messaging" campaigns which flood social media with repeated stock phrases while obfuscating their central originating source.

Of course, mining complete texts instead of quotations represents a significant scalability challenge. To address this, we propose an adaption of Kolak and Schilit (K&S)'s scalable architecture for finding "popular passages" in scanned books [11]. While their approach centered on use of the MapReduce paradigm for data-intensive, distibuted computing [7], they utilized Google's proprietary version of MapReduce and omitted some important practical details. We instead adopt the open-source Hadoop version of MapReduce [25] and discuss practical implementation issues and design patterns [14] for building scalable Hadoop systems for tasks such as ours. Because the K&S algorithm generates a quadratic data increase in what is already a data-intensive problem, requiring aggressive filtering, we describe an alternative approach which avoids this problem.

To validate our approach, we report on using our system to discover and browse memes in a 1.5 TB collection of 28 million crawled blogs. Our primary contributions include: 1) a novel approach and browser design for supporting critical literacy; and 2) a highly-scalable system architecture for meme discovery, providing a solid foundation for further system extensions and refinements. Our discussion of Hadoop algorithms and design patterns used should also be helpful

[1] `http://en.wikipedia.org/wiki/Critical_literacy`



Meme

"laura bush and cindy mccain"

▼ Other Mentions (Same Source)

2008: 09/02 09/03 09/06

▼ Other Mentions (General)

04/21/2008 www.perrspectives.com
09/03/2008 www.blogher.com
09/03/2008 d-day.blogspot.com

More

▼ Meme Related Phrases

laura bush and cindy mccain
16 mentions, 04/21/2008 - 09/25/2008

laura bush and cindy mccain's speech
8 mentions, 09/03/2008 - 09/18/2008

the first lady and cindy mccain
5 mentions, 09/04/2008 - 11/02/2008

▼ Memes of the Day

1. laura bush and cindy mccain
2. eight deaths were attributed to the storm in the
3. on this labor day weekend
4. the great threat to the obama biden

Tonight's line-up will feature a speech delivered via video by President George Bush, whose Monday night address was canceled due to the storm while Laura Bush and Cindy McCain stepped in. Also speaking will be Democratic senator and former vice president candidate Joe Lieberman, who is supporting presumptive GOP nominee John McCain over Democratic nominee Barack Obama. Former Tennessee Sen. Fred Thompson also will speak. While Backyard Conservative attends the event with videographer Leah Peterson, BlogHer will host a live-blog tonight by Chilihead.

The RNC held some programs Monday, including this leadership summit for young women videoblogged by Backyard Conservative for BlogHer. In addition, CNN reports that nearly 300 protesters were arrested after a march by about 5,000 people.

Other than Blogger Nice Deb, who is not a fan of protests, you wouldn't know about the protests from reading too many political blogs by women. Most are still talking about Alaska Gov. Sarah Palin, who is Sen. McCain's pick for vice president. On Monday Palin confirmed that her 17-year-old daughter is pregnant. In the past 24 hours, that discussion has evolved into an emotional conversation in the BlogHersphere about her personal decision to seek and hold elected office while being a mother of five children -- while other BlogHers question other peoples' willingness to judge Palin's decisions and still others ask

Feedback

Does the highlighted phrase reveal an interesting connection between the sources?

Figure 1: The **Meme Browser** makes readers aware of (discovered) underlying memes via highlighting. Navigation links let the reader explore detected memes in context, investigating connections they reveal between different documents. A *common phrase* (CP) is any phrase seen to occur in multiple documents, while a *meme* is a cluster a similar CPs. At all times, there exists an Active CP (ACP) and associated Active Meme (AM), for which relevant information is shown.

At top: The **Timeline Navigator** 1) indicates the date of the current document being read; 2) shows the temporal profile of mentions of the AM and 3) supports access and navigation to top memes on other days.

Below the timeline: The **Content Pane** 1) highlights a specific ACP for the AM in the context of the the current document; and 2) marks in gray any other highly-ranked memes present in the document. The reader may select a new AM at any time.

At left: The **Faceted Browsing Pane** provides information and navigation links relevant to the ACP and AM. From top-to-bottom, the pane shows 1) "Phrase": the ACP; 2) "Other Mentions (Same Source)": links to other mentions of the ACP by the same *source* (i.e. information provider); 3) "Other Mentions (General)": links to other mentions of the ACP by other sources; 4) "Meme Related Phrases": other CPs belonging to the AM; and 5) "Memes of the Day": an automatically-generated ranking of the the most informative memes for the current date.

At bottom: A **Feedback Widget** invites simple user feedback on meme quality to collect labels for training and evaluation.

for those more generally interested in data-intensive computing and effective use of Hadoop. Our source code[2] and *Meme Browser*[3] are both freely available online.

2. ENVISIONED USER EXPERIENCE

A screen shot of our *Meme Browser* interface is shown in Figure 1. A reader's experience with our system might start by opening a document and finding memes displayed, selecting a meme from opening the *Top Memes of the Day* page, or selecting some other date from the *Timeline Navigator*. Imagine the reader selects September 3, 2008 and its top meme, "laura bush and cindy mccain". The names might be familiar, but the reader might wonder why they are occurring together on this day. After selecting the meme, a representative document opens to a paragraph where the phrase is found highlighted:

> God loves Republicans because he allowed **Laura Bush and Cindy McCain** show how classy they were. Either of these women has more class in her little finger than Thunder Rodent Thighs has in her whole body.

At this point, the reader may become even more curious: what happened that day and why was the blogger so excited? He notices gray bars above the timeline indicating when the current meme was mentioned and its popularity in mentions over time. He then selects the first day where the peak is observed and finds a document with the following paragraph:

> Cindy McCain and first lady Laura Bush will appear before the Republican convention Monday to encourage people to donate to the relief efforts

[2]`https://bitbucket.org/softbass/meme-extractor`
[3]`http://odyssey.ischool.utexas.edu/mb/`

in the Gulf region, a senior McCain campaign official told reporters in a conference call.

Depending on the reader's familiarity with world events at that time, more questions might come to mind. "What happened in the Gulf area?"; "What did other politicians think about this?"; and "What was the public reaction?" The readers realizes that the highlighted phrases lead to other documents where the given phrase is mentioned. They see that "Hurricane Gustav slammed into the Louisiana coast," and "all political activity will be suspended for the time being."

Glancing at the *Meme Related Phrases* section, the reader can find additional phrases related to this same meme, along with how often they were mentioned and the time period in which they were most active. In this case, additional phrases include "laura bush and cindy mccain's speech" and "the first lady and cindy mccain", providing further points for exploration. Looking at the *Memes of the Day* section, the reader begins discerning relationships between other popular phrases and can continue to explore further.

3. SYSTEM ARCHITECTURE

Figure 2 presents our system's overall processing architecture. Initial preprocessing extracts and normalizes textual content (e.g. filtering out advertisements and HTML tags), as well as non-English documents and extremely short documents (see Section 4 for preprocessing details). Our subsequent processing follows Kolak and Schilit's high-level approach for finding "popular passages" in scanned books [11]. First, near-duplicate documents are identified and removed (Section 3.1). Next, we utilize a multi-stage, scalable MapReduce process to automatically find common phrases (CPs) that occur in multiple documents. We then rank CPs to identify the most informative ones to display to the user (Section 3.3). A clustering phase then groups similar CPs to form memes (Section 3.4). We report efficiency statistics in Section 4 and design of the *Meme Browser* in Section 5.

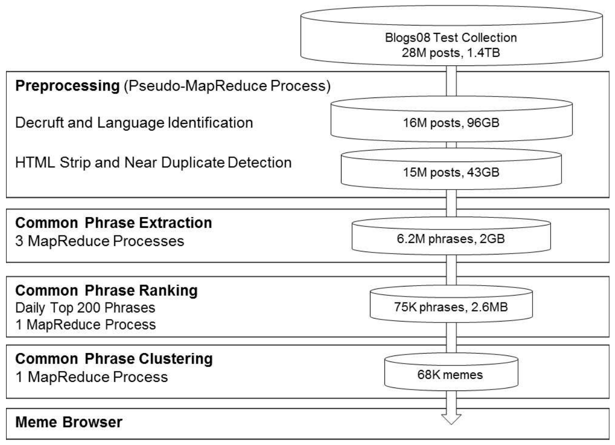


Figure 2: Our meme detection architecture involves preprocessing, near-duplicate detection, then finding, ranking, and clustering common phrases (CPs) to form memes.

3.1 Near-Duplicate Removal

Social media and news collections often contain many minor variants of the same document, e.g. different editors' versions of the same Associated Press story [1]. Because brute force near-duplicate document detection (NDD) requires $O(n^2)$ comparison of all document pairs, we developed several simple heuristics to reduce the search space and speed up comparison which served our immediate need. While secondary to our core work, we describe our approach here and refer the interested reader to other well-known NDD methods (cf. [23, 8]) and MapReduce approaches for performing efficient document comparison [13].

To reduce the number of comparisons required, we partition the document space to only compare documents which begin with the same character and have total word length within ± a parameter *window_size* of each other. To speed up the remaining comparisons, we: 1) reduce the vocabulary size by conflating all terms starting with the same four characters; 2) build a dictionary hashmap for mapping these (4-character) "terms" to numeric indices; 3) ignore term frequencies and record only binary presence vs. absence of each term; and 4) represent each document by a bit-vector for fast intersection operations. Note this representation of documents is used only for NDD.

Building the collection dictionary for the reduced vocabulary represents a minor variant on the cannonical `word count` example in the original MapReduce paper [7]. To reduce intermediate output, rather than emitting all tokens, we instead collected the document-specific vocabulary for each document in the Mapper and then output this vocabulary after the entire document has been processed (i.e. in-Mapper combiner pattern [14]). We also save all document-specific vocabulary from the Mapper (along with the document's length and first character) directly to HDFS. The hashmap is trivially built from the vocabulary and then shared with an *embarassingly parallel* processing stage which partitions the collection (i.e. the document-specific vocabulary for each document) across the cluster and uses the shared hashmap to convert each document to a bit vector.

Next, all documents with the same first character and word length are grouped together. The Mapper simply emits (character,length) as the primary key for each document, using the document ID as a secondary key. MapReduce secondary sorting [14] then groups documents by the primary key and sorts them by the secondary key. The reducer is just the identity function, outputting a sorted list of document IDs for each (character, length) primary key. A second hashmap is then trivially built for fast lookup.

Algorithm 1 details our NDD method. For each (character, length) bucket, we load the sorted list of document IDs plus document IDs in buckets within ±*window_size*. By symmetry, we need only consider longer lengths, as well as only compare documents with smaller IDs to those with larger. If document similarity is below the *dsim_threshold* parameter, the document with greater ID is identified as a near-duplicate and output for deletion.

As a slight variant to Jaccard similarity and Dice's coefficient, we compute document similarity by:

$$dsim(A,B) = \frac{|A \cap B|}{min(|A|,|B|)} \quad (1)$$

As an example, if one document has vocabulary [**a**, b, **f, g, h, i**] and another has [**a, f, g, h, i**, j, k, l], the resulting similarity is $\frac{5}{6}$. While this strict coefficient could be problematic were the document lengths considerably different, the parameter *window_size* prevents this case from occurring.

Algorithm 1 Near-Duplicate Document Removal

```
Parameter: $t \leftarrow dsim\_threshold$
Parameter: $\Delta \leftarrow window\_size$
Parameter: $map \leftarrow$ (character c, length l) $hashmap$
1: for each $(c, l)$ bucket $\in map$ do
2:     Sorted List $L \leftarrow \emptyset$
3:     for each $i \in [0 : \Delta]$ do
4:         $L.add$(all document IDs $\in map(c, l + i)$)
5:     for each $d \in L$ do
6:         for each $(d2.ID > d1.ID) \in L$ do
7:             if $dsim(d1, d2) > t$ then
8:                 $output(d2)$
9:                 $L.remove(d2)$
```

3.2 Finding Common Phrases

Our definition of *memes* was ideas represented by similar phrases that occur across multiple documents. Based upon the Kolak and Schilit (K&S) approach for finding "popular passages" in scanned books [11], we find identical phrases occurring in multiple documents, then group similar phrases into memes. While we adopt a similar 3-stage MapReduce architecture for finding the common phrases (CPs), our second and third stages differ markedly. As we shall further discuss, Stage 2 of the original K&S algorithm generates quadratic growth of data in an already data-intensive computation. While this growth can be mitigated by aggressive filtering, such reduction comes at the cost of increasing the false positive rate. Consequently, we describe an alternative formulation which finds CPs while avoiding this scalability tradeoff. We also replace the loosely-defined *grouping* phase of K&S with a well-specified method based on established clustering techniques (Section 3.4).

Algorithm 2 Stage 1: Shingle Table Generation

```
Parameter: $shingle\_size$
1: method MAP($doc\_id$, $text$)
2:     $position \leftarrow 0$
3:     for each $shingle\_size$ $shingle$ in $text$ do
4:         EMIT($shingle$, pair($doc\_id$, $position$))
5:         $position \leftarrow position + 1$

Parameter: $min\_count$
Parameter: $max\_count$                ▷ Maximum bucket size
1: method REDUCE($shingle$, $[(doc\_id, i)]$)
2:     $shingle\_count \leftarrow$ count($[(doc\_id, index)]$)
3:     if $(min\_count \leq shingle\_count \leq max\_count)$ then
4:         EMIT ($shingle$, $[(doc\_id, i)]$)
```

Stage 1: Shingle Table Generation. K&S begin by *shingling* each collection document, i.e. extracting sequential n-grams which overlap like roof shingles. For example, bigram shingling of "a man a plan" would yield three bigram shingles: "a man", "man a", and "a plan". For each observed shingle, a *bucket* is maintained tracking all (document ID, word position) pairs in which the shingle is seen to occur. The set of all observed shingles and their associated buckets defines the *shingle table.* Creating this table requires a single pass over the corpus, followed by a massive MapReduce sort for grouping bucket entries by shingle. To reduce the size of the shingle table, K&S describe two pruning optimizations: 1) discarding singleton shingles occurring only once in the collection (which cannot contribute to a match across multiple documents); 2) discarding very frequent shingles which would generate very large buckets (though the upper-limit they use is not specified). We adopt this shingle table generation process largely unmodified, although we do specify a more general lower-bound parameter for discarding rare shingles. A complete description of our shingle table generation process appears in Algorithm 2.

In the K&S approach, the bucket size upper-limit must be set aggressively to keep data small enough for practical processing in subsequent stages. While only the size of the output shingle table is affected at this stage, we shall describe the greater impact of bucket size on Stage 2 & 3.

As a final note on Stage 1 processing, an important point of Hadoop programming is not obvious from the canonical form of MapReduce pseudo-code shown in Algorithm 2: the Reducer does not actually load the entire bucket for a given shingle into memory. Instead, the Hadoop run-time provides an iteration mechanism to sequentially process each bucket entry one-at-a-time, avoiding this potential memory bottleneck. However, this raises the question of how we can implement an upper-limit constraint on bucket size shown in Line 2 of the Reducer.

The simplest, but memory-intensive, option is *Buffering*, which means retaining all bucket entries during iteration and only emitting entries after ensuring the upper-limit is not violated. The *Unbuffered* approach instead emits each bucket entry as we iterate but keeps count as we go. If the bucket size limit is ever violated, we record the shingle and ignore any subsequent bucket entries for it. After termination, we then go back and iterate over the list of recorded shingle IDs to prune them (and their associated buckets) from the shingle table. Such options are indicative of a common tradeoff in Hadoop programming. *Buffering* approaches are typically more efficient if the size of memory required is manageable; when data-intensive computing precludes this, however, we instead adopt an *Unbuffered* approaches. Between these extremes, one can alternatively specify a limited-size buffer and flush to disk whenever the buffer is filled. Since we assume a relatively large limit on bucket size, we do not buffer.

Algorithm 3 Stage 2: Grouping Shingles by Document

```
1: method MAP($shingle$, $[(doc\_id, i)]$)
2:     for each $(doc\_id, i)$ in $[(doc\_id, i)]$ do
3:         EMIT($doc\_id$, $(shingle, i)$)

1: method REDUCE($doc\_id$, $[(shingle, i)]$)      ▷ Identity
2:     EMIT ($doc\_id$, $[(shingle, i)]$)
```

Stage 2: Grouping Shingles by Document. Our approach to Stage 2 marks a significant departure from that of K&S; we simply perform a trivial a re-sort of the shingle table by document ID. In contrast, K&S output the shingle buckets for every shingle occurring in every document. To give a simple example, assume we have D documents which each contain S unique shingles, where each shingle occurs in some constant fraction of the D total documents (i.e. has bucket size $\frac{D}{k}$, for some k). For the $D * S$ shingles to be output, we must output a total of $\frac{D^2 * S}{k}$ bucket entries. This quadratic number of bucket entries to output as a function of the collection size D can be problematic for scaling the method to realistic collection sizes.

At this stage, the challenge is I/O intensive rather than memory intensive, impacting the amount of intermediate and persistent data which must be transferred over the network, buffered to/from disk, written persistently (end of Stage 2), then redistributed by HDFS over the network and read-in for Stage 3. As mentioned, K&S implicitly address this issue by aggressively restricting maximum bucket size. By using only positional information from the buckets for Stage 3 matching, they also cleverly avoid having to output shingle strings. Because we do not output buckets, we instead do output the shingle strings for Stage 3 matching.

In either approach, document shingles must be sorted by position for sequential processing in Stage 3. As before, we encounter another Hadoop space vs. time tradeoff. We could simply perform this sort in-memory with *Buffering*, or we could instead utilize slower *secondary sorting* to let the Hadoop run-time perform this sorting for us [25]. While large limit on bucket size led us to avoid buffering before, here we can buffer because documents are relatively short. A complete description of our Stage 2 appears in Algorithm 3.

Algorithm 4 Stage 3: Common Phrases (CP) Detection

```
Parameter: $max\_gap\_length$
 1: method MAP($doc\_id$, [($shingle$, $i$)])
 2:     $cp \leftarrow$ first $shingle$
 3:     $prev\_i \leftarrow$ first $i$
 4:     for each ($shingle$, $i$) in rest of ($shingle$, $i$) do
 5:         if $(i - prev\_i) \leq max\_gap\_length$ then
 6:             $start = end - (i - prev\_i)$
 7:             $cp \leftarrow cp + shingle_{start:shingle\_size}$
 8:         else
 9:             EMIT ($cp$, $doc\_id$)
10:             $cp \leftarrow shingle$
11:         $prev\_i \leftarrow i$
12:     EMIT ($cp$, $doc\_id$)

Parameter: $min\_doc\_count$
 1: method REDUCE($cp$, [$doc\_id$])
 2:     if (count([$doc\_id$]) $\geq min\_doc\_count$) then
 3:         EMIT ($cp$, [$doc\_id$])
```

Stage 3: Common Phrase (CP) Detection. Stage 3, shown in Algorithm 4, marks a complete departure from the K&S approach [11]. Whereas they find phrase matches based on shingle buckets, we use shingle strings instead for scalability. Moreover, without aggressive bucket size filtering, large bucket sizes become a problem of memory as well as I/O in their Stage 3 since large buckets lead to large numbers of active alignments buffered while iterating over document shingles . Finally, they give a "free pass" to any shingle pruned by the bucket size limit; because this limit is set aggressively to reduce quadratic data growth, an increasing number of spurious alignments will be made.

Our Mapper effectively just concatenates consecutive shingles, giving our "free passes" only to shingle gaps of length at most *max_gap_length*. Because scalability of our method allows us to adopt a conservative upper-limit on bucket size, we do not encounter many gaps from pruning of frequent shingles. At the other extreme, whereas K&S only prune singleton shingles in Stage 1, we prune rare shingles occurring less than *min_count* times (see Algorithm 2). However, singleton shingles always break phrases for K&S, even if the shingle is the only minor divergence from a longer alignment that ought to be captured. In contrast, our *max_gap_length* parameter allows us to preserve running phrases provided we do not miss too many shingles in a row, in which case such a gap is likely warranted anyway.

Whereas K&S find CPs by explicit sequence alignment across documents, we simply use the MapReduce sort phase to group together multiple occurrences of the same phrase. Because by default we find many CPs that are not interesting, we prune rare CPs (later ranking will further reduce this set). Buffering CPs in the reducer until the minimum threshold is met is trivial, after which we simply flush the buffer and emit all subsequent occurrences directly.

3.3 Common Phrase (CP) Ranking

As with the "popular passages" found by Kolak and Schilit (K&S) [11], the complete set of all CPs we find contains many phrases that are unlikely to be interesting to a reader. K&S perform ranking via a weighted geometric mean of length and frequency statistics. Our different social media data and usage context lead us to another formulation.

First and foremost, we are interested in developing a notion of information *source*, where multiple documents are agreed to originate from the same source. One challenge is conceptual: with news posts, for example, is the source an individual reporter, the reporter's local news organization, its umbrella corporation, or some other equivalence class we define over individuals or organizations. Another challenge is practical: how can we automatically infer this source (however source is defined), given the observable data? In this work, we make a simple assumption of treating each domain address as a unique source. For example, for a document with URL `http://copiouschatter.blogspot.com/2009/02/grumpy-old-men.html`, we take as source `copiouschatter.blogspot.com`. In general, a long tail of different URL naming patterns used by different social sites makes this a challenging mining problem [6].

For our context of usage, we want to rank some top set of CPs for each day to reveal to the reader. Unlike K&S, we do not use length statistics, but rather use a variant of TF-IDF style ranking. We use the number of unique sources the CP appears (S) and the number of documents (DF_{date}) for a given date, with an IDF-like effect from the number of documents across all dates in which the CP occurs (DF):

$$score_{date} = \frac{S \cdot DF_{date}}{DF} \tag{2}$$

Algorithm 5 illustrates our MapReduce procedure of extracting daily top k common phrases from CPs.

An interesting tradeoff to explore in future work is use of greater Map-side local aggregation to reduce intermediate data [14]. In particular, since we only want to output the top k values per day, this is an associative and commutative operation for which we could potentially filter in the Mapper without any approximation to correctness. The problem is that the Mapper here uses CPs as keys rather than dates, so the Mapper would be required to buffer data over all collection dates; more importantly, distribution of CPs for the same date across Mappers loses the critical locality needed. Another MapReduce process would be needed to re-sort the CPs by date, potentially negating any subsequent savings.

Algorithm 5 Rank top-k Common Phrases (CPs)

```
1: method MAP($cp$, [($doc\_id$, $source$)])
2:     Map[$date \mapsto source$] $sources \leftarrow \emptyset$
3:     Multiset $dates \leftarrow \emptyset$
4:     for each ($doc\_id$, $source$) in [($doc\_id$, $source$)] do
5:         $dates.add(date(doc\_id))$
6:         $sources[date(doc\_id)].add(source)$
7:     Set $uniqdates \leftarrow set(dates)$
8:     $DF \leftarrow |dates|$
9:     for each $date$ in $uniqdates$ do
10:        $S_{date} \leftarrow |sources[date]|$
11:        if $S_{date} > 1$ then
12:            $DF_{date} \leftarrow |dates[date]|$
13:            EMIT ($date, (cp, score_{date})$)

Parameter: $top\_k$
1: method REDUCE($date, [(cp, ms)]$)
2:     $ranked\_cps \leftarrow$ sort($[(cp, ms)]$ by $ms$)
3:     for each $i \in [1, top\_k]$ do
4:         EMIT ($date, ranked\_cps[i]$)
```

3.4 Common Phrase (CP) Clustering

Section 3.2 described how we find CPs at scale, while the previous section discussed how to filter out less interesting CPs. We now discuss how we cluster similar (unranked) CPs to form *memes*. Our approach here replaces the loosely-defined *grouping* phase of K&S with a well-specified method based on more standard clustering techniques.

In comparison to traditional clustering, our task is somewhat unusual. First, term vectors are far sparser vs. traditional document clustering since CPs are much shorter. Second, manual analysis suggests we are looking for many memes with few CPs, rather than a more traditional assigning of many examples to few clusters. We perform single-linkage hierarchical clustering [21] with cosine similarity:

$$cos(A, B) = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \quad (3)$$

Terms in vectors A and B are weighted by standard TF-IDF in which the weight w of a term t in CP p is given by

$$w_{tp} = TF_{tp} * \log \frac{N}{DF_t} \quad (4)$$

where TF denotes the frequency of term t in CP p, N denotes the total number of CPs, and DF denotes the number of CPs in which term t appears.

Like near-duplicate detection (Section 3.1), clustering is also naively an O(n^2) problem, involving similarity comparisons between all pairs of CP vectors. In this case, we adopt a standard information retrieval (IR) approach of using inverted indexing to efficiently restrict similarity comparisons to only those vectors which share one or more common terms. Efficient inverted indexing with Hadoop has been described in depth elsewhere [14]. We also use the standard IR method of stopwords to reduce vocabulary for further efficiency, though the set of stopwords is determined via a parameter $p\_threshold$, where the stoplist = $\{w | \frac{f_w}{N} < p\_threshold\}$. Crucially, stopwords are excluded from the index but not the CPs; if at least one non-stopword is shared between vectors, then similarity will be computed over the complete vocabulary. We refer to our approach as Indexed Hierarchical Clustering with MapReduce (IHCMR).

The Mapper algorithm is shown in Algorithm 6. Initialization begins by reading in all CPs (cp) from a shared file and the inverted index. The set of CPs is partitioned such that each node receives a subset of input IDs as input for Map invocations, while all CPs are considered as candidates (via the inverted index) to which the inputs CP may be compared. Due to symmetry, the input CP is only compared to candidate CPs with larger IDs. For each non-stopword in the input CP, we lookup the postings list in the inverted index and entries to the candidate set. Finally, the set of candidates are evalated. Only if the cosine similarity is greater than a parameter $sim\_threshold$ will a key-value pair of (sim, ($cp1$, $cp2$)) be emitted, where sim denotes the cosine similarity, and $cp1$ and $cp2$ denote the CPs compared. If a CP does not have any match above this threshold, then (0.0, $cp1$) is emitted instead (creating a singleton cluster).

Algorithm 6 Cluster Common Phrases (Mapper)

```
Parameter: $t \leftarrow similarity\_threshold$
1: $index \leftarrow$ read from disk                    ▷ Inverted Index
2: $cp \leftarrow$ read from disk          ▷ Vector of Common Phrases
3: method MAP($id$)                           ▷ Common Phrase ID
4:     Set $candidates \leftarrow \emptyset$
5:     for each $w \in (cp[id] \cap index)$ do
6:         for each ($cand\_id < id$) $\in index[w]$ do
7:             $candidates.add(cand\_id)$
8:     for each $cand\_id \in candidates$ do
9:         $sim \leftarrow cos(cp[id], cp[cand\_id])$          ▷ Cosine Sim.
10:        $emit\_flag \leftarrow False$
11:        if $sim > t$ then
12:            EMIT($sim$, ($cp[id], cp[cand\_id]$))
13:            $emit\_flag \leftarrow True$
14:    if $emit\_flag = False$ then
15:        EMIT(0.0, ($cp[id]$, null))
```

The Reducer, shown in Algorithm 7, is initialized with an empty cluster list. While the sort phase between Map and Reduce stages cannonically provides only "group-by" functionality (i.e. grouping together values with the same keys prior to invoking the Reducer), we utilize Hadoop's ability to sort the (floating point) keys in decreasing order. The Reducer is then iteratively invoked on decreasing similarity values, adding and merging clusters as appropriate. Final clusters are emitted at Reducer termination.

4. CASE STUDY: ANALYZING BLOGS08

This section reports a case study for using our system to discover memes in a large collection of crawled blogs. We begin by describing the dataset, computing infrastructure, parameter settings, and pre-processing. We then report empirical results of processing efficiency (time and space).

Dataset. We utilize the TREC[4] BLOGS08 collection[5] of Web Logs (blogs) crawled from January 14, 2008 to February 10, 2009. 1.3 million RSS feeds were polled weekly during

[4]`http://trec.nist.gov`

[5]`http://ir.dcs.gla.ac.uk/test_collections/blogs08info.html`

Algorithm 7 Cluster Common Phrases (Reducer)

```
1: $cl\_list \leftarrow ArrayList$
2: method REDUCE($sim, [(cp1, cp2)]$)
3:     if $sim = 0.0$ then
4:         Add $cp1$ to $cl\_list$ as a new cluster
5:     else
6:         if $cp1$ in $cl\_list$ and $cp2$ in $cl\_list$ then
7:             Merge clusters
8:         else if $cp1$ in $cl\_list$ then
9:             Add $cp2$ to cluster containing $cp1$
10:        else if $cp2$ in $cl\_list$ then
11:            Add $cp1$ to cluster containing $cp2$
12:        else
13:            Add new cluster($cp1$, $cp2$) to $cl\_list$

14: method CLOSE()
15:     EMIT($cl\_list$)
```

the period, and "permalink" documents (blog pages) were downloaded every two weeks. Each document includes all the components of a blog page as would be presented to a web browser. The collection consists of a total of 28,488,766 documents whose uncompressed size is 1445GB. Each document has a unique identifier and an associated crawl date. We have anecdotally observed differences of about 2-3 days between the crawl date and the posting date as indicated in the document text. At present, we simply use the crawl date for document dating; accuracy of dating could be further improved by either more frequent crawling or by extracting document dates from the text (e.g. via HeidelTime [22]).

Pre-processing. Because the BLOGS08 collection contains many non-English blog posts, advertisements, as well as HTML tags, we perform the following preprocessing to prepare documents for subsequent processing:

1. To extract meaningful content from a blog page (i.e. filtering out navigation links, advertisements, sidebar contents, and links to other sites), we apply Decruft, a Python implementation of ARC90's readability project[6].
2. To filter out non-English content, we utilize NLTK[7]'s language identification module.
3. For text analysis, HTML tags are removed with Beautiful Soup[8] and all text is converted to lowercase.
4. Short documents (fewer than five words) are removed
5. Documents are tokenized by words as follows (leading punctuation: ampersand, middle punctuation: back quote; final punctuation: apostrophe. "&" and ";" are included to preserve HTML entries such as "&amp;" or "&gt;." Aposrophe is used as a grammatical marker.

Compute Cluster. Experiments reported below were performed on a local Hadoop cluster at the Texas Advanced Computing Center[9] having 48 nodes, each having 8 2.53 GHz cores with 48GB RAM. When all 48 nodes are used, a total of 376 mappers and reducers can run simultaneously, using one Namenode and 47 Datanodes.

[6]http://www.minvolai.com/blog/decruft-arc90s-readability-in-python
[7]http://www.nltk.org
[8]http://www.crummy.com/software/BeautifulSoup
[9]http://www.tacc.utexas.edu

Parameter Settings. Table 1 lists our system parameters, the processing module in which each is used, and the parameter settings yielding the empirical results reported below. Current settings reflect heuristic tuning; our future work will investigate this parameter space in greater detail.

Process	Parameter	Value
Deduplication	$dsim\_threshold$	0.85
	$window\_size$	5
CP Finding: Stage 1	$shingle\_size$	5
	min_count	5
	max_count	225,000
CP Finding: Stage 3	min_doc_count	5
	max_gap_length	5
Ranking	top_k	200
Clustering	$p\_threshold$	0.01
	$sim\_threshold$	0.7

Table 1: System parameters and settings used.

Our parameters for $shingle\_size = 5$ and $min\_count = 5$ and $max\_count = 225,000$ differ from those of Kolak and Schilit (K&S) [11] (8, 1, and unspecified, respectively). While K&S were interested in finding long quotations in books, we are interested in shorter expressions such as "putting lipstick on a pig" and "the biggest financial fraud in history."

4.1 Processing Efficiency

Preprocessing. Initial filtering of extraneous page content and non-English posts significantly reduces collection size to 16,674,981 posts (totaling 96 GB). While HTML tags are filtered out for meme discovery, we do preserve tags for display in the Meme Browser (Section 5). HTML tag removal further reduces collection size to 47 GB.

Near-duplicate Detection (NDD). 1,577,525 documents (9.5%) were identified as near-duplicates, leaving 15,097,456 documents (43GB) after their removal. Unlike other experiments, NDD was performed early in our project using only a single 6-core server. With Hadoop pseudo-distributed mode, it required 6.5 days to complete. While this was sufficient for our proof-of-concept and could be further sped up via greater parallelism, our future work will instead pursue more efficient methods as discussed in Section 3.1.

Finding Common Phrases (CPs). Table 2 reports run time and volume of data generated by each MapReduce process in finding CPs. These results reflect use of 1 namenode and 17 datanodes, allowing 136 simultaneous mappers and reducers. A total of 5,631,742 CPs were found.

Stage	Mappers	Reducers	Time	Output
Stage 1	867	270	23:49	56GB
Stage 2	954	364	9:18	95GB
Stage 3	1786	364	4:03	2.0GB

Table 2: The total running time, number of mappers and reducers used, and size of output data generated by each MapReduce process in CP Finding.

Our lower-bound shingle frequency parameter $min\_count = 5$ for Stage 1 was quite aggressive, filtering out the vast majority of shingles observed (4,608,276,420, accounting for 98.27% of all shingles). In contrast, our upper-bound parameter of $max\_count = 225,000$ was very conservative and had very limited impact on performance. We plan to explore the tradeoff of more moderate settings in future work. After filtering, a total of 81,099,356 shingles remained. Stage 3 produces a total of 6,224,087 CPs.

Common Phrase (CP) Ranking. From the top 200 daily CPs, we gathered 75,039 unique phrases. The MapReduce process took 2 minutes and 51 seconds on a cluster of 18 nodes capable of 136 simultaneous mappers and reducers. Our job used 480 Mappers and 86 Reducers, and 6.2 million common phrases (2.0 GB) were processed.

Trial	Indexed	# Cores	Time
WEKA	-	1	> 96 hours
IHCMR	no	1	10:59:11
IHCMR	no	136	33:43
IHCMR	yes	1	16:20
IHCMR	yes	136	2:51

Table 3: Performance of alternative hierarchical clustering approaches for meme generation.

Common Phrase Clustering. Savings shown include:

1. Without indexing, our sparse vector representation over WEKA's off-the-shelf clustering using single core: from > 96 hours down to 11 hours
2. Indexed vs. unindexed: from 11 hours to 16 minutes (single core), from 34 to 3 minutes (136 cores)
3. Distributed vs. single core: from 11 hours to 34 minutes (unindexed) to 16 minutes to 3 minutes (indexed)

IHCMR is optimized to process extremely sparse data. The average length of the extracted common phrases is only 6.08 words while the vector space includes 28,694 words. When the dataset is converted into Attribute-Relation File Format (ARFF), 2.4MB of meme list data becomes 11GB in dense matrix ARFF, and 8.5MB in sparse matrix ARFF. We first tried to use WEKA COBWEB Hierarchical Clustering[10] with default settings. 2.5GB of memory was assigned to WEKA, and the clustering was run on a single processor at 2.67GHz. WEKA took more than 96 hours to finish the clustering. WEKA generates and processes the full dense matrix even from the sparse ARFF input, thus while WEKA compares the vectors of 28,694 dimensions, IHCMR compares the vectors of 6.08 dimensions in average.

Indexing and stopword removal. As discussed earlier, terms occurring more that 1% (*p_threshold*) of documents are not indexed (a total of 60 terms). Indexing and stopword removal reduce comparisons to only 411.78 per CP on average ($\sigma = 441.25$, range $[0 - 10,996]$ out of 75,034 phrases).

5. MEME BROWSER

As a first step toward our larger vision of enabling people to achieve greater critical literacy via technological assistance, we have designed a prototype *Meme Browser* for viewing and exploring detected memes in context. By visibly displaying underlying memes, readers become explicitly aware of their presence and have an opportunity to investigate them and the connections they reveal between sources. In addition to supporting critical literacy, the browser also allows readers to explore how such memes develop and propagate across sources and time. The browser serves both a casual reader, with limited time for independently finding reading many sources for connections, as well as a more motivated reader, who requires technological assistance to cope with the vast scale of diverse information sources today.

[10]`http://www.cs.waikato.ac.nz/ml/weka`

The Meme Browser (Figure 1) accesses collection documents via a `DocID` index, and an Apache-PHP-MySQL server provides database access to discovered memes. The browser interface is written in PHP with CSS, Javascript, and jQuery used for layout and dynamic presentation.

The Meme Browser's interface layout is informed by prior eye-tracking studies for faceted browsers and sponsored search results, which have shown typical heat-map distributions of user attention focused on top and left with exponential decay moving down and right [16]. Recall that a *common phrase* (CP) is any (unfiltered) phrase found in multiple documents, while a *meme* clusters similar CPs. The Browser's state maintains at all times an Active CP (ACP) and associated Active Meme (AM), for which relevant information is shown. We describe each component in detail below.

Timeline Navigator. At top, the Timeline Navigator 1) indicates the date of the current document being read; 2) shows the temporal profile of mentions of the AM and 3) supports access and navigation to top memes on other days.

If the mouse cursor hovers over the dateline, the date below the cursor is shown to assist in correct date selection. A click on the timeline reveals a pop-up display showing a *Top 10 Memes of* `<DATE>` listing, from which the user may then select a given meme. The selected meme then becomes the AM, a representative ACP for the AM selected, and a representative document for that day is opened.

Content Pane. Below the Timeline Navigator, the Content Pane 1) highlights a specific ACP for the AM in the context of the the current document; and 2) marks in gray any other highly-ranked memes present in the document. A new AM may be selected at any time.

The Content Pane displays the document (with its original formatting and layout) and any CPs found in it. The five most common CPs in the document are visibly marked to provide the user with multiple avenues for exploration. Any CPs other than the ACP are marked in a lighter gray color, which visibly indicates their presence while preserving the ACP as the primary focus. If the user has arrived at the document via a meme of CP-based navigation, the selected CP becomes the ACP and the document is automatically scrolled to vertically center the ACP and its surrounding context. If the user arrived at the document via traditional document-based navigation, then the document view opens at the beginning of the document. In this case, the first-occurring CP in the document is selected as the ACP, though it may not be visible until it is scrolled into view during the course of reading the document. If the user hovers the mouse cursor over the ACP, a pop-up window invites feedback on the ACP, similar to the *Feedback Widget* (see below).

Faceted Browsing Pane. Along the left side of the interface, the Faceted Browsing Pane provides information and navigation links relevant to the ACP and AM. From top to bottom, the following panes are provided:

- **Phrase** shows the Active Common Phrase (ACP).
- **Other Mentions (Same Source)** lists other (dated) documents from the same source which also contain the ACP. This allows the user to quickly assess the

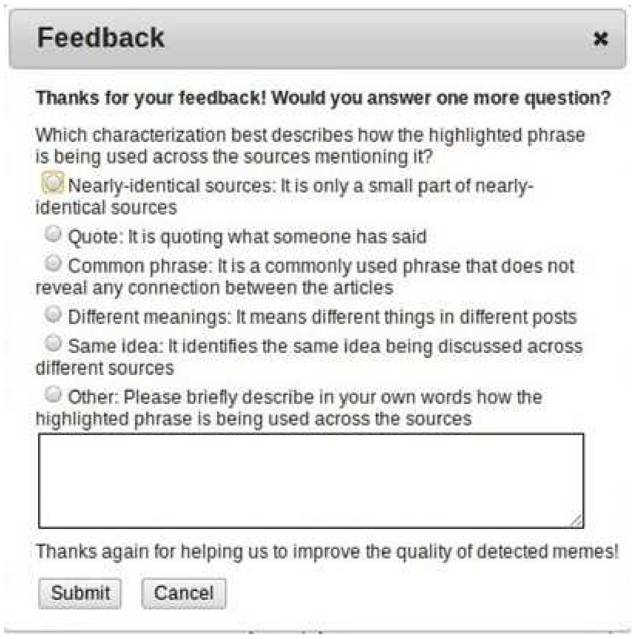
level of association between source and phrase. Navigation via this list demonstrates how the phrase context changes (or stays the same) over time.

- **Other Mentions (General)** takes a slightly broader approach, giving the user a list of other sources that also mention the ACP. Using this list, one can make several rough determinations of relationships across sources by looking at the dates in which they mention the same phrase. For example, if *Source A* mentions a meme and it consistently appears in the documents of *Source B* afterwards then the user may decide that the former "feeds" the latter information. Similarly, if different memes appear first in one source and then another, the relationship may be more bidirectional.
- **Meme Related Phrases** shows a different relationship than that between sources. Here, the focus is on the cluster of phrases itself. The selector displays a list of phrases that are similar to the ACP, along with when they first appear in the collection. Again, the user is able to quickly develop a rough idea of how the cluster of related phrases has evolved, particularly how the the meme text appeared and changed over time.
- **Memes of the Day** focuses less on relationships between sources and phrases and more on time and popularity. The selector shows a list of the 10 most frequently mentioned phrases on the same day of the viewed ACP. This list serves more as a "pulse" of the news by providing a snapshot of the topics that were trending on a given day. The user is able to jump to any of the most popular phrases and immediately see their relationships across sources and related phrases.

Feedback Widget. At bottom, a *Feedback Widget* invites user feedback on meme quality to collect labels for training and evaluation. It is located directly below the *Timeline Navigator* and above the *Content Pane*. Inspired by volunteer-based crowdsourcing approaches [3], it collects feedback on the quality and nature of discovered memes, generating "gold" labels to evaluate accuracy of discovered memes and training data for further improving this accuracy. A second benefit is making explicit to the reader that automation is imperfect, and that the quality and usefulness of detected memes should be critically assessed as should anything being read. The widget also promotes greater reader engagement through increased interaction.

To alert the reader to the Widget and invite their participation, a simple question asks, *Does the highlighted phrase reveal an interesting connection between the sources mentioning it?* Simple "Thumbs Up" and "Thumbs Down" icons fun, easy interaction. On clicking either button, a pop-up a window offers thanks and invites the reader to answer one further question (shown in Figure 3). Prior work has shown the benefit of designing for such staged feedback [10].

Because the definition and nature of memes requires careful explanation, the questions asked in Figure 3 reflect careful scrutiny of (correct and incorrect) memes identified by our system, as well as iterative revision, in order to provide annotators with clear guidelines for data labeling (e.g. to distinguish near-duplicate documents erroneous missed in earlier process vs. memes occurring in distinct documents). In addition to inviting readers to provide such feedback, our future work will also investigate use of Amazon Mechanical Turk to perform pay-based annotation work [18].



Feedback

Thanks for your feedback! Would you answer one more question?

Which characterization best describes how the highlighted phrase is being used across the sources mentioning it?

- [✓ Nearly-identical sources: It is only a small part of nearly-identical sources]
- Quote: It is quoting what someone has said
- Common phrase: It is a commonly used phrase that does not reveal any connection between the articles
- Different meanings: It means different things in different posts
- Same idea: It identifies the same idea being discussed across different sources
- Other: Please briefly describe in your own words how the highlighted phrase is being used across the sources

Thanks again for helping us to improve the quality of detected memes!

Figure 3: Whenever the Feedback Widget is used to provide yes/no feedback on meme quality, a pop-up dialog offers thanks and invites the reader to answer one further question.

6. RELATED WORK

Leskovec et al. [12] detect and analyze reuse of quotations online, but they do not address cases of text reuse beyond quotations (e.g. how shared experience of contemporary events can lead to to reuse of common phrases). Kolak and Schilit [11] mine full texts in entirety, but their approach was focused on scanned books, reflecting a different task and usage scenario. Plagiarism detection [4] is clearly related, though tends to focus on classifying documents rather than identifying and exploring examples of reuse. Somewhat farther afield, multiple sequence alignment methods from computational biology [17] may also be used to align variant phrases, though again our problem context and data differ.

Seo and Croft[20] detect local text reuse across newswire and blogs by breaking documents into sequences of words and measuring the frequency of shared phrases. Their DCT fingerprinting approach appears to be especially robust against small word variations within phrases, and they also exploit the idea that less common components are more important than higher frequency components. They show that DCT fingerprinting is both effective and efficient on blog data.

Kolak and Schilit [11]'s work, as our own, is made possible by the MapReduce framework was pioneered at Google in 2004 for effectively processing massive amounts of data using distributed computing, i.e. *data-intensive* computing [7]. MapReduce is designed to distribute data and computational tasks across a cluster of computers that all work simultaneously. Google's MapReduce architecture uses its proprietary distributed GoogleFS filesystem to replicate data in chunks across multiple computers and distribute it for locality of computation, bringing computation to the data as much as possible. The model is ideal for some forms of computation which can be divided into individual units that operate largely independently, unlike alternative message-passing approaches supporting greater synchronization.

Hadoop is an open-source Java implementation of the MapReduce programming model that operates on the Hadoop Distributed File System (HDFS) [25]. Inspired by similar work at Google, Hadoop offers developers a framework for parallel computation of large-scale data analysis. As an open-source system, Hadoop has gained tremendous popularity across industry and research environments in recent years, especially as "big data" grows even bigger and more common every day. As more researchers and practioners have begun working with Hadoop, there has been increased interest in developing and disseminating effective design patterns for efficient and easier data-intensive programming [14].

7. CONCLUSION AND FUTURE WORK

The question of how to effectively find relevant information has driven decades of information retrieval (IR) research. Relatively less attention has been directed toward helping people to more easily and effectively contextualize, interpret, assess information once it is found. The scale of information overload today threatens effective sense-making as well search. More research is needed to develop effective technologies for helping us better analyze and evaluate what we read, e.g. to support critical decision making.

As a modest step in this direction, we utilized MapReduce (Hadoop) and IR methods to identify and rank informative memes in a large-scale blog collection. These memes can then be browsed and explored in context via our *Meme Browser*. In future work, we are particularly interested in tracking and letting the user interactively explore how similar phrases in a meme cluster evolve over time, as well as showing the individuals and communities involved in disseminating, altering, and responding to memes. Readers should be able to see how sources follow others in using the same meme, or memes that commonly appear together. By recognizing inter-connection of sources through common meme flow patterns, we can become more critical readers.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers and Luis Francisco-Revilla for their valuable feedback. Prateek Maheshwari contributed to our literature review, and Hang Yu contributed to our preliminary work. This work was partially supported by Science and Technology Foundation of Portugal (FCT) grant UTA Est/MAI/0006/2009, a John P. Commons fellowship, a Longhorn Innovation Fund for Technology (LIFT) award, and Amazon Web Services. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their institution or funding agencies.

9. REFERENCES

[1] R. Barzilay and L. Lee. Learning to paraphrase: An unsupervised approach using multiple-sequence alignment. In *Proc. NAACL HLT*, pages 16–23, 2003.

[2] M. Bernstein. On Hypertext Narrative. In *HyperText*, pages 5–14, 2009.

[3] A. Brew, D. Greene, and P. Cunningham. Using crowdsourcing and active learning to track sentiment in online media. In *ECAI 2010*, pages 145–150, 2010.

[4] S. Eissen and B. Stein. Intrinsic plagiarism detection. *Advances in Info. Retrieval*, pages 565–569, 2006.

[5] R. Ennals, B. Trushkowsky, and J. Agosta. Highlighting disputed claims on the web. In *Proc. of WWW*, pages 341–350, 2010.

[6] G. Forman, E. Kirshenbaum, and S. Rajaram. A novel traffic analysis for identifying search fields in the long tail of web sites. In *WWW*, pages 361–370, 2010.

[7] S. Ghemawat and J. Dean. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI*, pages 137–149, 2004.

[8] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. of VLDB 1999*, pages 518–529, 1999.

[9] M. A. Hearst. Emerging trends in search user interfaces. In *Proc. HyperText*, pages 5–6, 2011.

[10] R. Kohavi, R. Longbotham, D. Sommerfield, and R. Henne. Controlled experiments on the web: survey and practical guide. *Data Mining and Knowledge Discovery*, 18(1):140–181, 2009.

[11] O. Kolak and B. N. Schilit. Generating links by mining quotations. *HyperText*, pages 117–126, 2008.

[12] J. Leskovec, L. Backstrom, and J. Kleinberg. Meme-tracking and the dynamics of the news cycle. *Proc. of ACM SIGKDD '09*, page 497, 2009.

[13] J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *Proceedings of ACM SIGIR*, pages 155–162, 2009.

[14] J. Lin and C. Dyer. Data-intensive text processing with mapreduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.

[15] S. Moulthrop. What The Geeks Know: Hypertext and the problem of literacy. In *Proceedings of ACM HyperText*, pages 227–231, 2005.

[16] V. Navalpakkam, J. Rao, and M. Slaney. Using gaze patterns to study and predict reading struggles due to distraction. In *ACM CHI*, pages 1705–1710, 2011.

[17] C. Notredame. Recent evolutions of multiple sequence alignment algorithms. *PLoS computational biology*, 3(8):1405–1408, Aug. 2007.

[18] H. Ryu and M. Lease. Crowdworker Filtering with Support Vector Machine. In *Proc. ASIS&T*, 2011.

[19] B. N. Schilit and O. Kolak. Exploring a digital library through key ideas. In *JCDL*, pages 177–186, 2008.

[20] J. Seo and W. Croft. Local text reuse detection. In *Proc. of ACM SIGIR*, pages 571–578. ACM, 2008.

[21] R. Sibson. SLINK: an optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, 1973.

[22] J. Strötgen and M. Gertz. HeidelTime: High quality rule-based extraction and normalization of temporal expressions. In *SemEval*, pages 321–324, July 2010.

[23] M. Theobald, J. Siddharth, and A. Paepcke. Spotsigs: robust and efficient near duplicate detection in large web collections. In *SIGIR*, pages 563–570, 2008.

[24] J. Valenza. Web 2.0 meets information fluency: Evaluating blogs, 2010. January 20. `http://21cif.com/rkitp/assessment/v1n5/valenza1.5_blogeval.html`.

[25] T. White. *Hadoop: The Definitive Guide.* O'Reilly Media, Inc., 2nd edition, 2010.