A Fuzzy, Incremental, Hierachical Approach of Clustering Huge Collections of Web Documents

Patrick Hennig[1], Philipp Berger[1], Christian Godde[2], Daniel Hoffmann[2] and Christoph Meinel[3]
Hasso-Plattner-Institut
University of Potsdam, Germany
[1]Email: {patrick.hennig, philipp.berger}@hpi.uni-potsdam.de
[2]Email: {christian.godde, daniel.hoffmann}@student.hpi.uni-potsdam.de
[2]Email: office-meinel@hpi.uni-potsdam.de

Abstract— *Since every day millions of posts are published inside the blogosphere a huge collection of web documents develops. Clustering this ever-changing collection is a very time consuming task. Therefore some certain challenges has to be accomplished because a clustering cannot be executed from scratch all the time. The presented fuzzy, incremental and hierarchical clustering algorithm tries to succeed these challenges with both, clustering terms and documents in a meaningful way and keep them up-to-date all the time. Furthermore we take a critical look at the performance, which is crucial on such a live data collection.*

Keywords: Web Mining, Data Mining, Blog Mining, Clustering, Blogs, Unstructured Data

1. Introduction

With a tremendous circulation of several hundred million blogs worldwide, the ever changing collection of weblogs are getting bigger and bigger every day. For mining, modeling and presenting this think tank of open-source intelligence a very intelligent and fast clustering method is needed. This incorporates some very important and as well very difficult challenges.

Since a clustering of an ever-changing huge corpus consumes a lot of computing power, it cannot be calculated from scratch every time a new document is added. Therefore the first challenge is to keep the clustering up-to-date. In addition, the number of clusters cannot be fixed in the beginning, since different granularity levels should be covered. As a consequence, it has to be a hierarchical clustering. Furthermore, since we know a lot about the ambiguity from semantic web it is not appropriate to force documents and terms to fit into exactly one strict cluster. This means, the clustering algorithm has to consider fuzziness.

In the next Section the overall project, the presented work is integrated, is explained in more detail before some related work for such clustering techniques are introduced. In Section 5, we describe the basic challenges for the presented clustering. Afterwards the main algorithm and its characteristics is described in more detail.

In addition, the presented algorithm gets evaluated in Section 7. Furthermore the performance is evaluated and decisions are explained in order to improve the performance of the presented algorithm. Finally, some future work that can be conducted is highlighted before the paper is summarized in the last Section.

2. Project Scope

With a wide circulation of more than 200 million *weblogs* worldwide, weblogs with good reason are one of the most important data streams in the World Wide Web. Therefore, weblogs offer access to latest information discussed in the real world. Since writing posts in weblogs goes along with a high editorial effort, the available information is of major interest. However, for a user it is becoming harder and harder to gain an overview of all discussions in the blogosphere. It is almost impossible for a user to extract information from the web, especially from the blogosphere. Hence, a system that collects information from the blogosphere and presents it to the user in a very meaningful way would be of great use.

Therefore, mining, analyzing, modeling and presenting this enormous amount of data is the overall aim of the project the presented work is integrated in. This enables the user to detect technical trends, political climates or news articles about a specific topic. Most approaches to mining and analyzing such a huge amount of data focus on offline algorithms which use pre-aggregated results. This is in contrast to the continuously growing nature of the World Wide Web. As a result, including the latest data is one of the key aspects of data mining on the web. This is exactly the topic covered by the *BlogIntelligence*[1] project. Since BlogIntelligence usees different text mining analytics a clustering is a fundamental factor of success.

[1]BlogIntelligence is a tool the extract and analyze data such as content and links from weblogs of the German blogosphere in order to visualize this information and provide a tool to explore and discover the world of social media. More information on: `http://www.blog-intelligence.com/`

3. Related Work

As the field of cluster analysis is very complicated and algorithms heavily depend on the specific domain, there are as many different papers on clustering as there are possible use-cases. In this section we mention some of them that are closely related to our own work.

The initial idea for an incremental clustering was introduced by Arnaud Ribert et al. [1]. They explain an efficient way of inserting new elements into existing hierarchical clusterings and give a brief overview of hierarchical clustering itself. In their evaluation they show that the required memory and number of computations can be significantly reduced with an incremental algorithm.

Among other improvements in our algorithm we mainly focused on the efficiency of the distance matrix calculations. The map reduce technique that is now in use and described in the implementation section is based on the work of Elsayed et al. [2].

With the objective that the algorithm should perform in reasonable time the focus moves towards building up a hierarchy tree. Therefore we looked at different approaches for preparing the data in order to improve efficiency of clustering. Dash et al. [3] relies on partitioning the items beforehand.

The 'BIRCH' algorithm by Zhang et al. [4] would be an alternative that uses a different tree structure for a first rough clustering. It can be computed in quadratic time and thereby reduces the time that has to be spent for the final clustering. Unfortunately the time complexity even for these sophisticated techniques can not be better than O(n^2).

4. Clustering Techniques

a) Partitional Clustering: The most prominent representative of the first class is the k-means algorithm. Partitional clusters divide a database into a specific amount of clusters. In general this number of clusters (k) has to be given in advance and it is not possible to change it later on. The problem is to find the optimal k if the exact composition of the data is not well known before.

b) Hierarchical Clustering: A Hierarchical clustering does not produce a partition into a specified number of clusters. It produces aso called dendrogram. This dendogram can be build either top-down or bottom-up. The first type is called divisive clustering. Starting with the whole data set and splitting it into two subsets until every item is represented by a leaf node. The top-down clustering is called agglomerative clustering. Starting with each item in a separate node and combining two nodes until the root includes the whole data set.

In both cases, when finished, the dendrogram forms a hierarchical binary tree with each item of the given database as leaf node and the root representing the whole database. Every node in the tree represents a subset of the database and thereby a sub-cluster. The greater the difference between the two sub-nodes, the higher the node is in the hierarchy. Knowing this, we can get few big clusters as well as many small clusters from the same dendrogram without recomputing the clustering. The biggest disadvantage of hierarchical clustering versus partitional cluster techniques is the higher computational time.

c) Hard and Fuzzy Clustering: In general, each item is assigned to exactly one cluster it fits best (e.g. the average distance to the other cluster members is the smallest). This is called hard clustering. If we want to express that an item can be quite similar to different other items we have to introduce a fuzzy clustering. In fuzzy clustering techniques the membership of an item to a cluster is expressed as a probability value which in total sums up to 1. That way any item can appear in multiple clusters.

d) Incremental Clustering: Incremental clustering has to be taken into account either if the corpus to be clustered is too big to keep the information about all items at the same time in memory or if the corpus is increasing over time and we do not want to repeat the clustering every time a new item is added. Incremental clustering has the advantage that we can add the items of the database one by one and therefore reduce time and space complexity. A huge disadvantage is the order-dependence of the resulting clustering. We might get completely different results if the order of the items added to the clusters is changed. Thus, it is very difficult to guarantee the quality of the clustering.

5. Clustering the Blogosphere

The goal of the system that is described in this paper is to improve and add new functionality to *BlogIntelligence*. Inside the blogosphere cluster analysis faces a lot of old and new problems. The first question to be answered is: Where do we want to find clusters? This question can be answered in two ways. On the one hand, blogs and their post can be clustered. These represent documents in a more classical view on cluster analysis. So the goal is to find groups of blogs that might cover the same topic like politics or computer science. This is important because it helps finding new links between different blogs and thereby helps authors and users to explore the Web 2.0.

On the other hand, we are also interested in clustering features of blogs and posts. These are terms or also tags of blog posts. By clustering terms it gets necessary to find words that together best describe specific topics. So we are able to categorize blogs even better and again improve the possibility to explore the blogosphere. These term clusters can be used for further analysis like trend detection and visualization.

The main challenge the blogosphere imposes on clustering is its size. The user might dive into the blogosphere from the top and narrow the search down to parts of the blogosphere he is interested in. Hence, the clustering should provide a way of getting a very rough division as well as a very detailed one.

The size of the blogosphere also pushes the need for the efficiency of the clustering algorithm. Another characteristic of the blogosphere is that new documents are added continuously. Because of this, it is not applicable to constantly recompute the clustering. It has to be possible to change the existing one if a new item has to be integrated.

All in all, a system is needed that clusters both, documents and terms, that is capable of adding new documents to an existing cluster, that returns clusters of variable size and amounts, and that assigns terms and documents to multiple topics and groups. This leads to a hierarchical, incremental, fuzzy clustering.

6. Implementation

6.1 General Idea

Before the main algorithm is described in detail we first have to define the most important terms concerning cluster analysis within blog posts.

6.1.1 Post term matrix

The post term matrix describes the distribution of terms across all blog posts. For every post a list contains the frequency of each possible term. The optimal case provides normalized tf-idf values as described in the next Section. The other way around, it is also possible to get a list that for a specific term containing all frequencies among all blog posts. The number of terms defines the dimensionality of a document vector and the other way round. Typically, the post term matrix is sparse because for each blog post only the most representing terms (terms with the highest tfidf) are stored.

6.1.2 Term frequency - inverse document frequency

The *tf-idf* value [5] describes the frequency of a term in a document in comparison to its frequency in the whole corpus. A term that appears in all available documents is not very descriptive for a single document and therefore has a low *tf-idf* score for every document. On the other hand, a term that only occurs in one document has a high score for this specific document even if its frequency is low and therefore has a high meaning for this document. The general formula for calculating the tf-idf is given as follows:

$$tfidf(t_i, d_j) = tf(t_i, d_j) * \log \frac{N}{n_i}$$

$tf(t_i, d_j)$ is the frequency of term t_i in document d_j. N is the number of documents in the corpus. And n_i is the number of documents that contain term t_i.

6.1.3 Similarity measure

To measure the similarity between documents and terms, we use the cosine similarity. It measures the inner angle between two tf-idf vectors.

$$sim(d_1, d_2) = \frac{d_1 \cdot d_2}{|d_1||d_2|}$$

To accelerate similarity computation normalized tf-idf vectors can be used. That way the cosine similarity becomes a simple inner product

$$sim(d_1, d_2) = d_1 \cdot d_2 = \sum_{i=0}^{n} w(t_i, d_1) w(t_i, d_2)$$

where $w(t, d)$ is the tf-idf of term t in document d.

6.1.4 Initial Clustering

Initially a common agglomerative clustering algorithm is used. To determine how many documents should be taken into account at the initial clustering, we take a look at *Heaps' law* [6]. This helps to reduce the computational effort enormously. Heaps' law describes how the size of the vocabulary V_R grows depending on the document size n:

$$V_R(n) = Kn^{\beta}$$

For English language normally K is between 10 and 100 and β is between 0.4 and 0.6. The vertical line in 1 shows a possible cut to cover sufficient terms of the existing corpus. This ensures that most terms are covered in the cluster and the clustering runs still in an acceptable time.

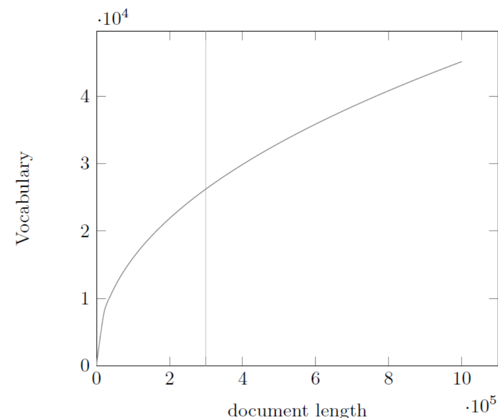

Fig. 1: Example plot for Heaps law $V_R(n) = Kn^{\beta}$

6.2 Approaching Fuzzyness

Since a term can appear in several contexts, a fuzzy clustering should be used as well. Thus a term can be contained in several clusters. To achieve this, we search for all pairs of items that have a similarity above a certain threshold. These pairs constitute the leaves in the clustering tree. In

order to reduce the computational effort, the calculation of the similarity can be limited to terms that have been occurred with each other in the same document once.

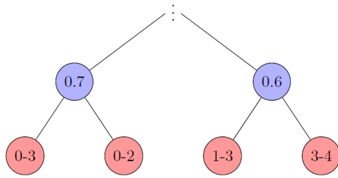

Fig. 2: Two clusters: $\{0, 2, 3\}$ and $\{1, 3, 4\}$

6.3 Algorithm

The initial clustering algorithm consists of three steps: finding the items that have a high similarity and associate them in leaves, adjusting the distance matrix and computing the hierarchical clustering.

1) **Finding leaves** For efficient similarity calculation a MapReduce [7] approach fits to our needs. MapReduce is a framework that allows parallel execution of an algorithm on several CPUs or machines. The *map* functions map a list of key-value pairs from one domain to another, the *shuffle* function groups the results and the *reduce* function computes the results for each group (see Figure 3).

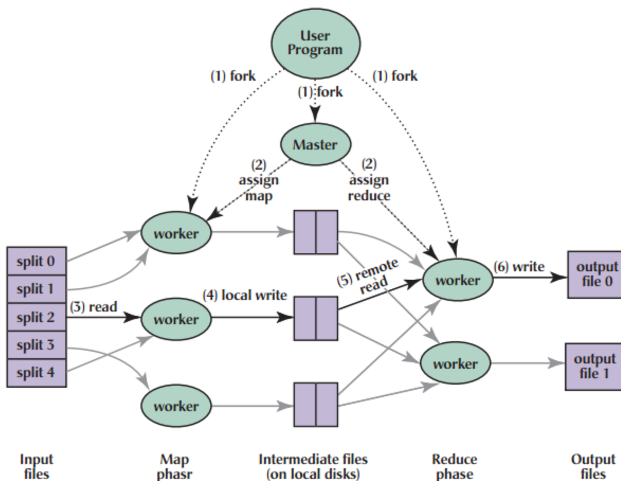

Fig. 3: General MapReduce architecture (taken from [7])

This method is used for computing document similarity with the approach described in [8]. For each term a list of the documents it appears in and its tf-idf value in this document is constructed. The *map* task multiplies the term weights for each pair of documents in the posting. The output is sorted by the keys, resulting in a list of pairs of documents as keys and a list of products computed in the *map* step. *Reduce* sums up the products and outputs the similarity values for the pairs of documents (see Figure 4). While this method is much faster than calculating the cosine similarity by iterating over the vectors, it uses more memory. Nevertheless, since the MapReduce approach allows simple distributed computing and main memory gets cheap we favor the faster version.

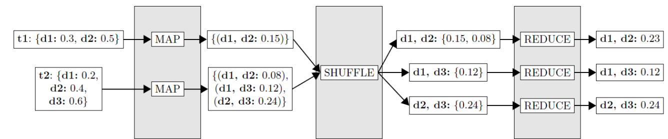

Fig. 4: Computing document similarities with MapReduce

After computing the similarity values, each pair having a similarity above a certain threshold is associated in a leaf node. The tf-idf vector for this pair is the average of the two tf-idf vectors of the associated items.

2) **Calculating the distance matrix** To compute a matrix of distances between these new leaves, we use the similar *map* and *reduce* function mentioned before with the new tf-idf vectors.

3) **Hierarchical Clustering** The implementation makes use of the *Efficient HAC* [9, 368] (Algorithm 1), which uses priority queues. For each row of the distance matrix a priority queue keeps the entries sorted in decreasing order. The maximum value of $P[k]$ consists of the similarity and index of the cluster most similar to the cluster with index k.
Since the C++ standard library priority queue does not allow erasing other elements than the top, we implemented a suitable version using the *std::make_heap*, *std::push_heap* and *std::pop_heap* functions. In this algorithm I stores the clusters that are still active, A stores the clustering as a sequence of merges. With these optimization the time complexity becomes $O(n^2 \log n)$.

6.4 Adding and removing documents/terms

6.4.1 Adding

To add an element into the cluster tree it is compared with each existing element as shown in Algorithm 2. If the similarity is above the threshold mentioned in Section 6.3, a new leaf is constructed with this pair of elements. For each new leaf its place in the clustering hierarchy is determined by starting at the root node and following the path with the highest similarity. To accelerate this procedure, each centroid of a cluster is cached. If a leaf is reached, a new inner node with this leaf and the new leaf as children is inserted. Its centroid is computed and the similarity values up to the root are refreshed.

Input: $d_1, \ldots, d_n$
for *n = 1 to N* **do**
  for *i = 1 to N* **do**
    C[n][i].sim = $d_n \cdot d_i$;
    C[n][i].index = i;
  end
  I[n] = 1;
  P[n] = priority queue for C[n] sorted on sim;
  P[n].DELETE(C[n][n]) (don't want self-similarities);
end
A = [ ];
for *k = 1 to N - 1* **do**
  k1 = arg maxk: I[k]=1 P[k].MAX().sim;
  k2 = P[k1].MAX().index;
  A.APPEND(k1, k2);
  I[k2] = 0;
  P[k1] = [ ];
  forall the *i with I[i] = 1 and i = k1* **do**
    P[i].DELETE(C[i][k1]);
    P[i].DELETE(C[i][k2]);
    C[i][k1].sim = SIM(i, k1, k2);
    P[i].INSERT(C[i][k1]);
    C [k1][i].sim = SIM(i, k1, k2);
    P[k1].INSERT (C[k1][i]);
  end
end
return *A*;

Algorithm 1: Efficient HAC algorithm using priority queues (from [9, 386])

Input: element e' to add
foreach *other element e* **do**
  if *similarity(e', e) > threshold* **then**
    construct new leaf(e', e);
  end
end
foreach *new leaf l'* **do**
  node = root;
  while *!node.isLeaf()* **do**
    node = $\arg\max_{n \in \{node.left, node.right\}} sim(l', n)$;
  end
  construct new inner node(leaf, node);
end
refresh similarity up to root;

Algorithm 2: Add element to clustering tree

6.4.2 Removing

When removing an element from the tree each leaf containing this element is inspected as shown in Algorithm 3. In order to not remove an element that is just contained in one leaf, we need to keep track of the number of leaves an element appears in. Suppose we want to delete it from the tree. If a leaf (e, e') is the only one containing the element e' that is not deleted, e is deleted from the node. Otherwise the father node of the node to be deleted is replaced by its sibling node. Finally, the similarities of the nodes up in the hierarchy get refreshed.

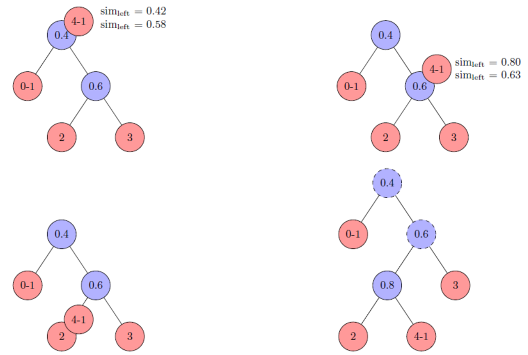

Fig. 5: Inserting a new leaf into the cluster tree, following the path of highest similarity. The similarity values in the dashed nodes needs to be refreshed.

Input: element e to delete
foreach *leaves (e,e') containing e* **do**
  if *number of leaves containing e' > 1 or the leaf contains only one element* **then**
    replace father node with brother node ;
  else
    delete element from leaf;
  end
  refresh similarity up to root;
end

Algorithm 3: Remove element from clustering tree

6.5 Differences between Document and Term Clustering

When we talk about adding a new element to the cluster, regarding our use-case, this means generally adding a new blog post and thereby a new document to the corpus. Adding a new term independently is rather impossible. This leads to the following observation.

While clustering documents, the insertion of a new item does not raise any complications. Table 1 shows a simple document-term matrix after document 3 was added to. Because the new document does not change the correlations of all other documents, the new document can simply be compared to the existing documents and a new base nodes can be added accordingly.

When adding a document, there might also be new terms, but the frequencies for these in the existing documents will always be zero. Otherwise the term must have already

	T_0	T_1	T_2
Document 1	1	0	0
Document 2	0	1	0
Document 3	0.5	0	0.5

Table 1: Doc-Term Matrix after adding the new Document 3

been in the corpus and is not new. So the assumption that existing correlations do not change, still holds even if a new document brings up new terms.

On the other hand, while clustering terms some new challenges has to be faced. As shown in Table 1, the terms T_0 and T_2 get more similar because they are now used together in the same document. The same way a new document might as well decrease the similarity of two existing terms. Taking this into account we adjust our algorithm for adding elements as follows:

For all terms in the new document

1) find all base nodes that include the term as one of their elements
2) remove all those base nodes
3) find and add new base node according to Section 6.4

This ensures that changes in the increasing or decreasing similarity between existing items are propagated to the cluster without recomputing the clustering as a whole. Nevertheless the number of nodes that has to be removed and added can be quite huge, depending on the composition of the clustering and the new document. As a side-effect the cluster tree might get smaller although a new document is added.

6.6 Runtime versus Correctness

The presented method for adding new elements does not always produce a correct cluster tree, but it is much faster than computing a new cluster tree. Additionally, the clustering is very heavily order-dependent as already pointed out. Therefore the more documents are added incrementally the more inaccurate the cluster becomes. To maintain an at least proximate correct partitioning, the clustering algorithm can be performed in background periodically. And the currently active cluster can be replaced by the latest computed one.

7. Evaluation

7.1 Improvements Through Parallelization

For parallelization, OpenMP [10] allows simple multi-platform parallel programming. Compiler directives indicate the code parts that are executed in parallel. *#pragma omp parallel* denotes a parallelized section. Then, as many threads are created as there are CPUs. OpenMP takes over the whole work of thread creation and management. *#pragma omp for* distributes the iterations of a for loop to these threads. Critical sections are marked with *#pragma omp critical section*. If a compiler does not understand these directives, the code still compiles and the for loop is executed sequentially.

The part that profits the most of parallelization is the pair-wise similarity computation which is used for example for creating the distance matrix. By using map reduce as pointed out in Section 6.3, a high degree of parallelization can be achieved. Parallelizing the construction of the dendogram has to be done carefully because we are looking for the global most similar items and not for the one in a subset of the dataset. This requires a high amount of inter-thread communication.

7.2 Runtime

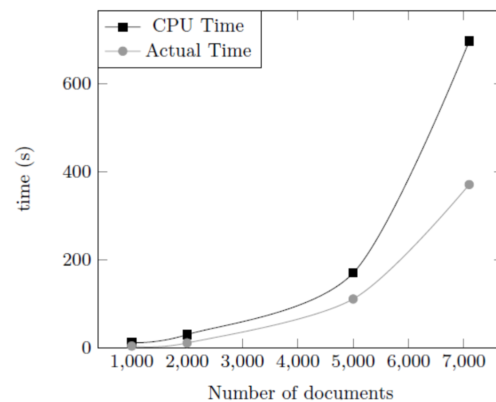

Fig. 6: Runtime of the initial clustering algorithm for different corpus sizes

For evaluation purposes we used the Classic collection of documents called Classic3 published by Cornell University[1]. This collection of documents is often used as benchmark in text mining. It consists of 7095 documents and 5896 terms.

Figure 6 shows the results of the presented clustering algorithm with this collection. It ran on a 4-core 64bit Linux-System with 8GB main memory. The similarity threshold for combining elements into one base node was set to 0.8 which resulted in approximately 25% combined base nodes and 75% single base nodes while clustering documents. The CPU Time is the aggregated time of all cores working while the actual time is the time the program actually runs. The results show that the presented algorithm has a complexity of about $O(n^3)$. Which is the general complexity of hierarchical clustering algorithms. However these figures do not entirely represent the proposed algorithm. The documents in the classic collection have the characteristic that documents with a lower ID also only use terms with lower IDs. This leads to considerably shorter term vectors for smaller corpus sizes

[1]You can find more information about the document collection at `http://www.dataminingresearch.com/index.php/2010/09/classic3-classic4-datasets/` and download it at `ftp://ftp.cs.cornell.edu/pub/smart/`

and longer vectors for the whole corpus. So the numbers are not unbiased and in practice the complexity is better than $O(n^3)$.

8. Future Work

a) Memory versus Time Complexity: A big step in the direction of reasonable performance was the decision to use map reduce for pairwise similarity computation. This reduced the calculation time of the distance matrix by 80%. Tthe downside of this is the fact that our implementation of the map reduce tasks extremely increases the memory consumption. This is due to the fact that during the map step all values in the document-term matrix are emitted before they are combined during the reduce step. However, as the whole environment is based on main memory computation and runs on systems offering a lot of main memory the advantage of better performance outweighs this disadvantage.

b) Distributed Hierarchical Clustering: In Section 7.1 we pointed out that parallelizing the hierarchy construction is not easy because every time time all distance values are needed. Dash et al. [3] introduced a way to compute the hierarchy in a distributed manner called 'partially overlapping partitioning (POP)'. The algorithm is split into two parts. At first the data is distributed into p partitions with p given by the number of processors available and clustered into sub-clusters. In the second phase the p sub-clusters have to be combined and further clustered. This approach can be combined with the approach presented in this paper to reduce the execution time.

9. Conclusion

With this paper we presented an approach of a fuzzy, hierarchical, incremental clustering. The initial step of finding items that have a high similarity and combining them into one base node enables a fuzzy clustering. Hereby, terms representing different topics can be grouped together.

The hierarchical characteristic provides the user with different levels of granularity for the composition of weblogs. By incrementally adding new documents we can cover the fact that bloggers publishes new blog posts all the time. As we showed, the performance of the presented algorithm is already pretty good, but can be boosted with some smaller improvements. Finally, the algorithm is able to deal with all challenges that are imposed by the overall use-case of clustering the blogosphere.

References

[1] A. Ribert, A. Ennaji, and Y. Lecourtier, "An incremental hierarchical clustering," in *IAPR-VI'99*, Trois Rivières, Québec, Canada, 1999, pp. 586–591.

[2] T. Elsayed, J. Lin, and D. W. Oard, "Pairwise document similarity in large collections with mapreduce," 1999, pp. 265–268.

[3] M. Dash and P. Scheuermann, "Efficient parallel hierarchical clustering," in *In International Europar Conference (EURO-PARŠ04*, 2004.

[4] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: an efficient data clustering method for very large databases," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '96. New York, NY, USA: ACM, 1996, pp. 103–114. [Online]. Available: http://doi.acm.org/10.1145/233269.233324

[5] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval," pp. 132–142, Dec. 1988.

[6] H. S. Heaps, *Information Retrieval: Computational and Theoretical Aspects*. Orlando, FL, USA: Academic Press, Inc., 1978.

[7] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[8] T. Elsayed, J. Lin, and D. W. Oard, "Pairwise document similarity in large collections with mapreduce," in *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*. Association for Computational Linguistics, 2008, pp. 265–268.

[9] C. D. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[10] OpenMP Architecture Review Board. (2011) OpenMP application program interface version 3.1. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP3.1.pdf