Scaling Out All Pairs Similarity Search with MapReduce

Regular Paper

Gianmarco De Francisci Morales
IMT Institute for Advanced Studies
Lucca, Italy
gianmarco.dfmorales@imtlucca.it

Claudio Lucchese
ISTI-CNR
Pisa, Italy
claudio.lucchese@isti.cnr.it

Ranieri Baraglia
ISTI-CNR
Pisa, Italy
ranieri.baraglia@isti.cnr.it

ABSTRACT

Given a collection of objects, the All Pairs Similarity Search problem involves discovering all those pairs of objects whose similarity is above a certain threshold. In this paper we focus on document collections which are characterized by a sparseness that allows effective pruning strategies.

Our contribution is a new parallel algorithm within the MapReduce framework. The proposed algorithm is based on the inverted index approach and incorporates state-of-the-art pruning techniques. This is the first work that explores the feasibility of index pruning in a MapReduce algorithm. We evaluate several heuristics aimed at reducing the communication costs and the load imbalance. The resulting algorithm gives exact results up to 5x faster than the current best known solution that employs MapReduce.

1. INTRODUCTION

The task of discovering similar objects within a given collection is common to many real world applications and machine learning problems. To mention a few, recommendation and near duplicate detection are a typical examples.

Item-based and user-based recommendation algorithms require to find, respectively, similar objects to those of interest to the user, or other users with similar tastes. Due the the number of users and objects present in recommender systems, e.g. Amazon, similarity scores are usually computed off-line.

Near duplicate detection is commonly performed as a preprocessing step before building a document index. It may be used to detect redundant documents, which can therefore be removed, or it may be a hint for spam websites exploiting content repurposing strategies. Near duplicate detection finds application also in the area of copyright protection as a tool for discovering plagiarism, for both text and multimedia content.

In this paper, we focus on document collections. The reason is that documents are a particular kind of data that exhibits a significant sparseness: only a small subset of the whole lexicon occurs in any given document. This sparsity allows to exploit indexing strategies that reduce the potentially quadratic number of candidate pairs to evaluate.

Furthermore, we are interested in discovering only those pairs of documents with high similarity. If two documents are not similar, they usually do not contribute to any of the applications we mentioned above. By setting a minimum similarity threshold, we can also embed aggressive pruning strategies.

Finally, the size of the collection at hand poses new interesting challenges. This is particularly relevant for Web-related collections, where the number of documents involved is measured in billions. This implies an enormous number of potential candidates.

More formally, we address the all pair similarity search problem applied to a collection $\mathcal{D}$ of documents. Let $\mathcal{L}$ be the lexicon of the collection. Each document d is represented as a $|\mathcal{L}|$-dimensional vector, where $d[i]$ denotes the number of occurrences of the i-th term in the document d. We adopt the cosine distance to measure the similarity between two documents. Cosine distance is a commutative function, such that $\cos(d_i, d_j) = \cos(d_j, d_i)$.

Definition 1. *Given a collection $\mathcal{D} = \{d_1, \dots, d_N\}$ of documents, and a minimum similarity threshold σ, the* All Pairs Similarity (APS) *problem requires to discover all those document pairs $d_i, d_j \in \mathcal{D}$, such that:*

$$\cos(d_i, d_j) = \frac{\sum_{0 \le t < |\mathcal{L}|} d_i[t] \cdot d_j[t]}{\|d_i\| \|d_j\|} \ge \sigma$$

We normalize vectors to unit-magnitude. In this special case, the cosine distance becomes simply the dot product between the two vectors, denoted as $\text{dot}(d_i, d_j)$.

The main contribution of this work is a new distributed algorithm that embeds state-of-the-art pruning techniques. The algorithm is designed within the MapReduce framework, with the aim of exploiting the aggregated computing and storage capabilities of large clusters.

The rest of this paper is organized as follows: in Section 2 we introduce a few concepts needed for the description of the proposed algorithm. We also describe the two most relevant contributions to our work. Section 3 incrementally describes our proposed algorithm and heuristics, highlighting the strengths and weaknesses for each strategy. Section 4 presents the results of our experimental evaluation. Finally, in Section 5 we summarize our contribution and present some ideas for future work.

2. BACKGROUND

A serial solution. The most efficient *serial* solution to the APS problem was introduced in [3]. The authors use an *inverted index* of the document collection to compute similarities. An inverted index stores an *inverted list* for each term of the lexicon, i.e. the list of documents containing it, together with the weight of the term in each document. More formally, the inverted list of the term t is defined as $I_t = \{\langle d_i, d_i[t]\rangle | d_i[t] > 0\}$.

It is evident that two documents with non-zero similarity must occur in the same inverted list at least once. Therefore, given a document d_i, by processing all the inverted lists I_t such that $d_i[t] > 0$, we can detect all those documents d_j that have at least one term in common with d_i, and therefore similarity greater than 0.

This is analogous to information retrieval systems, where a query is submitted to the inverted index to retrieve matching/*similar* documents. In this case, a full document is used as a query.

Actually, index construction is performed incrementally, and simultaneously to the search process. The matching and indexing phase are performed one after the other. The current document is first used as a query to the current index. Then it is indexed, and it will be taken into account to answer subsequent document similarity queries. Each document is thus matched only against its predecessors, and input documents can be discarded once indexed.

Usually the matching phase dominates the computation because its complexity is quadratic with respect to the length of the inverted lists. In order to speed-up the search process, various techniques to prune the index have been proposed [2, 3].

We focus on the first technique proposed in [3]. Let $\hat{d}$ be an artificial document such that $\hat{d}[i] = \max_{d\in\mathcal{D}} d[i]$. The document $\hat{d}$ is an upper-bounding pivot: given a document d_i, if $\cos(d_i, \hat{d}) < \sigma$ then there is no document $d_j \in \mathcal{D}$ being sufficiently similar to d_i. This special document $\hat{d}$ is exploited as follows.

Before indexing the current document d_i, the largest b such that $\sum_{0\le t<b} d_i[t]\cdot\hat{d}[t] < \sigma$ is computed. The terms $t < b$ of a document are stored in a remainder collection named $\mathcal{D}_R$, and only the terms $t \ge b$ of the current document are inserted into the inverted index. The pruned index provides partial scores upon similarity queries.

The authors prove that for each document d_i currently being matched, their algorithm correctly generates all the candidate pairs (d_i, d_j) using only the indexed components of each d_j. For such documents the remainder portion of d_j is retrieved from $\mathcal{D}_R$ to compute the final similarity score.

Finally, the authors propose to leverage the possibility of reordering the terms in the lexicon. By sorting the terms in each document by frequency in descending order, such that $d[0]$ refers to the most frequent term, most of the pruning will involve the longest lists.

A parallel solution.
When dealing with large datasets, e.g. collections of Web documents, the costs of serial solutions are still not acceptable. Furthermore, the index structure can easily outgrow the available memory. The authors of [5] propose a parallel distributed solution based on the MapReduce framework [4].

MapReduce is a distributed computing paradigm inspired by concepts of functional languages. More specifically, MapReduce is based on two higher order functions: Map and Reduce. The Map function applies a User Defined Function (UDF) to each key-value pair in the input, which is treated as a list of independent records. The result is a second list of intermediate key-value pairs. This list is sorted and grouped by key, and used as input to the Reduce function. The Reduce function applies a second UDF to every intermediate key with all its associated values to produce the final result.

The signatures of the functions that compose the phases of a MapReduce computation are as follows:

$$\begin{aligned} \mathit{Map}: &\quad [\langle k_1, v_1\rangle] &\rightarrow&\quad [\langle k_2, v_2\rangle] \\ \mathit{Reduce}: &\quad \{k_2 : [v_2]\} &\rightarrow&\quad [\langle k_3, v_3\rangle] \end{aligned}$$

where curly braces "{ }″ square brackets "[ ]" and angle brackets "⟨ ⟩" indicate respectively a map/dictionary, a list and a tuple.

The Map and Reduce function are purely functional and thus without side effects. For this reason they are easily parallelizable. Fault tolerance is easily achieved by just re-executing the failed function. MapReduce has become an effective tool for the development of large-scale applications running on thousand of machines, especially with the release of the open source implementation Hadoop [1].

Hadoop is an open source MapReduce implementation written in Java. Hadoop provides also a distributed file system called HDFS, that is used as a source and sink for MapReduce executions. HDFS deamons run on the same machines that run the computations. Data is split among the nodes and stored on local disks. Great emphasis is placed on data locality: the scheduler tries to run mappers (task executing the Map function) on the same nodes that hold the input data. This helps to reduce network traffic.

Mappers sort and write intermediate values on the local disk. Each reducer (task executing the Reduce function) pulls the data from various remote disks. Intermediate key-value pairs are already partitioned and sorted by key by the mappers, so the reducer just merge-sorts the different partitions to bring the same keys together. This phase is called *shuffle*, and is the most expensive in terms of I/O operations. The MapReduce data flow is illustrarted in Figure 1.

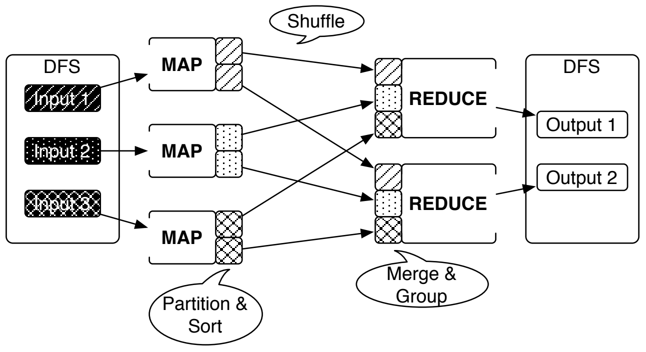

Figure 1: Data flow in a MapReduce job

Since building and querying incrementally a single shared index in parallel is not a scalable solution, a two phase algorithm is proposed in [5]. In the first phase an inverted index of the collection is built (indexing phase), and in the second phase the similarity score is computed directly from the index (similarity phase). Each phase is implemented as a MapReduce execution.

We describe this algorithm in more detail in the following section. The algorithm is used throughout the paper as a baseline for the evaluation of our proposed solutions. Indeed, the authors of [5] propose an algorithm for computing the similarity of every pair of documents. For this reason, we add a final filtering phase that discards the documents that do not satisfy the threshold.

3. ALGORITHM

In this section we describe the algorithm used to solve the APS problem using the MapReduce framework. We start from a basic algorithm and propose variations to reduce its cost. The main idea we try to exploit is that many of the pairs are not above the similarity threshold, so they can be pruned early from the computation. This fact is already exploited in state-of-the-art serial algorithms [2, 3]. Our goal is to embed these techniques into the MapReduce parallel framework.

3.1 Indexed Approach (Version 0)

A simple solution to the pairwise document similarity problem [5] can be expressed as two separate MapReduce jobs:

1. Indexing: for each term in the document, the mapper emits the term as the key, and a tuple consisting of document ID and weight as the value, i.e. the tuple $\langle d, d[t] \rangle$. The MapReduce runtime automatically handles the grouping by key of these tuples. The reducer then writes them to disk to generate the inverted lists.

2. Similarity: for each inverted list I_t, the mapper emits pairs of document IDs that are in the same list as keys. There will be $m \times (m-1)/2$ pairs where $m = |I_t|$ is the inverted list length. The mapper will associate to each pair the product of the corresponding term weights. Each value represents a single term's contribution to the final similarity score. The MapReduce runtime sorts and groups the tuples and then the reducer sums all the partial similarity scores for a pair to generate the final similarity score.

This approach is very easy to understand and implement, but suffers from various problems. First, it generates and evaluates all the pairs that have one feature in common, even if only a small fraction of them are actually above the similarity threshold. Second, the load is not evenly distributed.

The reducers of the similarity phase can only start after all the mappers have completed. The time to process the longest inverted list dominates the pair generation performed by the mappers. With real-world data, which follows a Zipfian or Power-law distribution, this means that the reducers usually have to wait for a single mapper to complete. This problem is exacerbated by the quadratic nature of the problem: a list twice as long takes about four times more to be processed.

A document frequency cut has been proposed to help reducing the number of candidate pairs [5]. This technique removes the 1% most frequent terms from the computation. The rationale behind this choice is that because these terms are frequent, they do not help in discerning documents. The main drawback of this approach is that the resulting similarity score is not exact.

3.2 Pruning (Version 1)

To address the issues in the previous approach, we employ the pruning technique described in Section 2. As a result, during the indexing phase, a smaller pruned index is produced. On the one hand, this reduces the number of candidate pairs produced, and therefore the volume of data handled during the MapReduce shuffle. On the other hand, by sorting terms by their frequency, the pruning significantly shortens the longest inverted lists. This decreases the cost of producing a quadratic number of pairs from these lists.

This pruning technique yields correct results when used in conjunction with dynamic index building. However, it also works when the index is built fully before matching, and only the index is used to generate candidate pairs. To prove this, we show that this approach generates every document pair with similarity above the threshold.

Let d_i, d_j be two documents and let b_i, b_j be, respectively, the first indexed features for each document. b_i and b_j are the boundaries between the pruned and indexed part as shown in Figure 2. Without losing generality, let $b_j \succ b_i$ (recall that features are sorted in decreasing order of frequency, so b_j is less frequent than b_i). We can compute the similarity score as the sum of two parts:

$$\mathrm{dot}(d_i, d_j) = \sum_{0 \le t < b_j} d_i[t] \cdot d_j[t] + \sum_{b_j \le t < |\mathcal{L}|} d_i[t] \cdot d_j[t]$$

While indexing, we keep an upper bound on the similarity between the document and the rest of the input. This means that $\forall \dot{d} \in \mathcal{D}$, $\sum_{0 \le t < b_j} \dot{d}[t] \cdot d_j[t] < \sigma$. Thus, if the two documents are above the similarity threshold $\mathrm{dot}(d_i, d_j) \ge \sigma$, then it must be that $\sum_{b_j \le t < |\mathcal{L}|} d_i[t] \cdot d_j[t] > 0$. If this is the case, then $\exists\, t \succ b_j \,|\, (d_i \in I_t \wedge d_j \in I_t)$. Therefore, our strategy will generate the pair (d_i, d_j) when scanning list I_t.

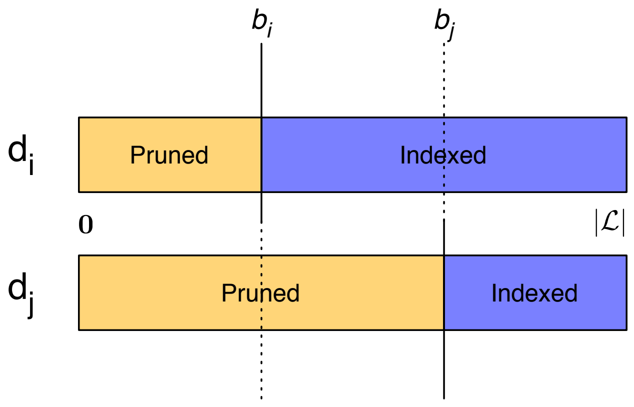

Figure 2: Pruned Document Pair: the left part (orange/light) has been pruned, the right part (blue/dark) has been indexed.

The Reduce function in the similarity phase receives a reduced number of candidate pairs, and computes a partial similarity score. Due to index pruning, no partial scores will be produced from the inverted lists $\{I_t \,|\, 0 \le t < b_j\}$, since these inverted lists will not contain both documents. Therefore, the reducer will have to retrieve the original documents, and compute the contribution up to term b_j in order to produce the exact similarity score.

We chose to distribute the input locally on every node[1]. The performance penalty of distributing the input collection is acceptable for a small number of nodes, but can become a bottleneck for large clusters. Furthermore, the input is usually too big to be kept in memory, so we still have to perform 2 random disk I/O per pair.

Finally, to improve the load balancing we employ a simple bucketing technique. During the indexing phase, we randomly hash the inverted lists to different buckets. This spreads the longest lists uniformly among the buckets. Each bucket will be consumed by a different mapper in the similarity phase. While more sophisticated strategies are possible, we found that this one works well enough in practice.

3.3 Flagging (Version 2)

In order to avoid the distribution of the full document collection, we propose a less aggressive pruning strategy. Our second approach consists in flagging the index items instead of pruning them. At the same time, the flagged parts of the documents are written as a side effect file by the mappers of the indexing phase. This *"remainders"* file is then distributed to all the nodes, and made available to the reducers of the similarity phase. The remainders file is normally just a fraction of the size of the original input (typically 10%), so distributing it is not a problem. During pair generation in the similarity phase, a pair is emitted only if at least one of the two index items is not flagged.

Our pair generation strategy emits all the pairs for the features from b_i to $|\mathcal{L}|$, so we just need to add the dot product of the remainders. The remainders file is small enough to be easily loaded in memory in one pass during the setup of the reducer. Thus, for each pair we only need to perform two in-memory lookups and compute their dot product. This process involves no I/O, so it is faster than the previous version.

The main drawback of this version is that it generates more pairs than version 1. This leads to unnecessary evaluation of pairs and consequently to wasted effort.

3.4 Using Secondary Sort (Version 3)

This version tries to achieve the benefits of both previous versions. Observe that for every pair (d_i, d_j) one of the two documents has been pruned up to a term that precedes the other (remember that features are sorted according to their frequency). Let this document be the LPD (Least Pruned Document) of the pair. Let the other document be the MPD (Most Pruned Document). In Figure 2, d_i is the LPD and d_j is the MPD.

We use version 1 pair generation strategy and version 2 remainder distribution and loading. To generate the partial scores we lack (from 0 to b_j), we just need to perform the dot product between the whole LPD and the remainder of the MPD. The catch is to have access to the whole LPD without doing random disk I/O and without keeping the input in memory.

Our proposed solution is to shuffle the input together with the generated pairs and route the documents where they are needed. In order to do that, we employ Hadoop's Secondary Sort feature. Normally, MapReduce sorts the intermediate records by key before starting the reduce phase. Using secondary sort we ask Hadoop to sort the records also by a secondary key while the grouping of values is still performed only by primary key. Instead of using the whole pair as a key, we use the LPD as the primary key and the MPD as the secondary key.

As a result, input values for the reducer are grouped by LPD, and sorted by both LPD and MPD, so that partial scores that belong to the same pair are adjacent. The LPD document from the original input that we shuffled together with the pairs is in the same group. In addition, we impose the LPD document itself to sort before every other pair using a fake minimum secondary key. This allows us to have access to the document before iterating over the values, and therefore to perform the dot products on the fly. This is a representation of the input for the reduce of the similarity phase:

$$\underbrace{\langle \mathbf{d_i} \rangle; \langle (d_i, d_j), W_{ij}^A \rangle; \langle (d_i, d_j), W_{ij}^B \rangle; \langle (d_i, d_k), W_{ik}^A \rangle; \ldots}_{group\ by\ key\ d_i}$$

$$\underbrace{\langle \mathbf{d_j} \rangle; \langle (d_j, d_k), W_{jk}^A \rangle; \langle (d_j, d_k), W_{jk}^B \rangle; \langle (d_j, d_l), W_{jl}^A \rangle; \ldots}_{group\ by\ key\ d_j}$$

First, we load the document $\mathbf{d_i}$ in memory. Then, for each stripe of equal consecutive pairs (d_i, d_j), we sum the partial scores W_{ij}^X for each common term X. Finally, we compute the dot product between the LPD and the remainder of the MPD, which is already loaded in memory from the remainders file. We repeat this cycle until there are no more values. After that we can discard the LPD from memory and proceed to the next key-values group.

4. EXPERIMENTAL RESULTS

In this section we describe the performance evaluation of the proposed algorithms. We ran the experiments on a 5-node cluster. Each node is equipped with two Intel Xeon E5520 CPUs clocked at 2.27GHz. Each CPU features 4 cores and Hyper-Threading for a total of 40 virtual cores. Each node has a 2TiB disk, 8GiB of RAM, and Gigabit Ethernet.

On each node, we installed Ubuntu 9.10 Karmic, 64-bit server edition, Sun JVM 1.6.0_20 HotSpot 64-bit server, and Hadoop 0.20.1 from Cloudera (CDH2).

We used one of the nodes to run Hadoop's master daemons (Namenode and JobTracker), and the rest were configured as slaves running Datanode and TaskTracker daemons. Two of the cores on each slave machine where reserved to run the daemons, the rest were equally split among map and reduce slots (7 each), for a total of 28 slots for each phase.

We tuned Hadoop's configuration in the following way: we allocated 1GiB of memory to each daemon and 400MiB to each task, we changed the HDFS block size to 256MiB and the file buffer size to 128KiB. We also disabled speculative execution and enabled JVM reuse and map output compression.

For each algorithm, we wrote an appropriate combiner to reduce the shuffle size (a combiner is a reduce-like function that runs inside the mapper to aggregate partial results). In our case, the combiners perform the sums of partial scores in the values, according to the same logic used in the reducer. We also implemented raw comparators for every key value used in the algorithms in order to get better performance (raw comparators are used to compare keys during sorting without deserializing them into objects).

[1]using Hadoop's Distributed Cache feature

# documents	17,024				30,683				63,126			
# terms	183,467				297,227				580,915			
# all pairs	289,816,576				941,446,489				3,984,891,876			
# similar pairs	94,220				138,816				189,969			
algorithm version	v0	v1	v2	v3	v0	v1	v2	v3	v0	v1	v2	v3
# evaluated pairs (M)	109	65	82	65	346	224	272	224	1,519	1,035	1,241	1,035
# partial scores (M)	838	401	541	401	2,992	1,588	2,042	1,588	12,724	6,879	8,845	6,879
index size (MB)	46.5	40.9	46.5	40.9	91.8	82.1	91.7	82.1	188.6	170.3	188.6	170.3
remainder size (MB)			4.7	4.7			8.2	8.2			15.6	15.6
running time (s)	3,211	1,080	625	554	12,796	4,692	3,114	2,519	61,798	24,124	17,231	12,296
avg. map time (s)	413	197	272	177	2,091	1,000	1,321	855	10,183	5,702	7,615	5,309
stdv. map time (%)	137.35	33.53	34.97	25.74	122.18	31.52	34.08	35.27	129.65	24.52	30.27	24.43
avg. reduce time (s)	57	558	35	79	380	2,210	191	220	1,499	11,330	1,112	1,036
stdv. reduce time (%)	18.76	5.79	13.59	14.66	48.00	5.62	23.56	14.61	13.55	2.46	8.37	9.51

Table 1: Statistics for various versions of the algorithm

We used different subsets of the TREC WT10G Web corpus. The dataset has 1,692,096 english language documents. The size of the entire uncompressed collection is around 10GiB.

We performed a preprocessing step to prepare the data for analysis. We parsed the dataset, removed stopwords, performed stemming and vectorization of the input. We extracted the lexicon and the maximum weight for each term. We also sorted the features inside each document in decreasing order of document frequency, as required by the pruning strategy.

4.1 Running Time

We evaluated the running time of the different algorithm versions while increasing the dataset size. For all the algorithms, the indexing phase took always less than 1 minute in the worst case. Thus we do not report indexing times, but only similarity computation times, which dominate the whole computation.

We set the number of mappers to 50 and the number of reducers to 28, so that the mappers finish in two waves and all the reducers can run at the same time and start copying and sorting the partial results while mappers are still running. For all the experiments, we set the similarity threshold to 0.9.

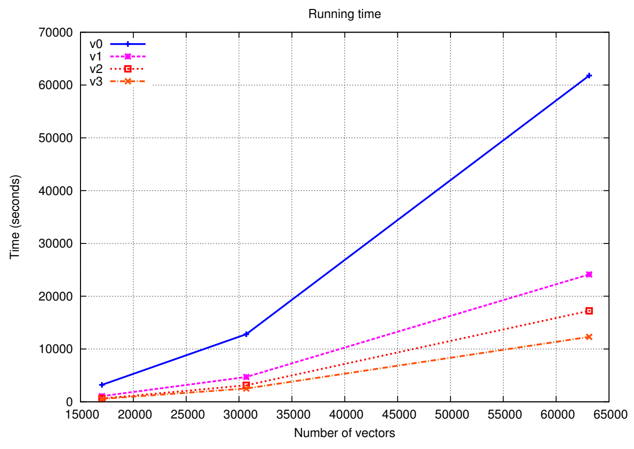

Figure 3: Computation times for different algorithm versions with varying input sizes

Figure 3 shows the comparison between running times for the different algorithms. The algorithms are all still quadratic, so doubling the size of the input roughly multiplies by 4 the running time. All the advanced versions outperform the basic indexed approach. This can easily be explained once we take into accounts the effects of of the pruning and bucketing techniques we applied.

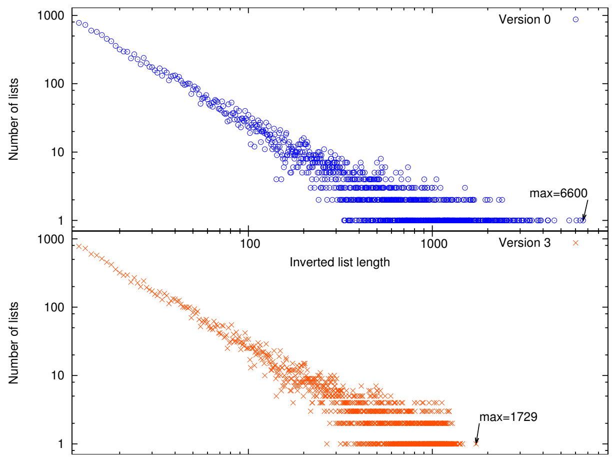

Figure 4: Index size distribution with and without pruning

Figure 4 shows the effects of pruning. The maximum length of the inverted lists is drastically reduced in version 3 compared to version 0. This explains their different running times, as the algorithm is dominated by the traversal of the longest inverted list. Figure 5 shows the effects of bucketing. The load is evenly spread across all the mappers, so that the time wasted waiting for the slowest mapper is minimized. It is evident also from Table 1 that the standard deviation of map running times is much lower when bucketing is enabled.

On the largest input, version 3 is 5x faster than version 0, 2x faster than version 1 and 1.4x faster than version 2. This is caused by the fact that version 3 does not access the disk randomly like version 1 and evaluates less pairs than version 2. Exact times are reported in Table 1.

Version 3 outperforms all the others in almost all aspects. The number of evaluated pairs and the number of partial scores are the lowest, together with version 1. Version 3 has also the lowest average map times. The standard deviation

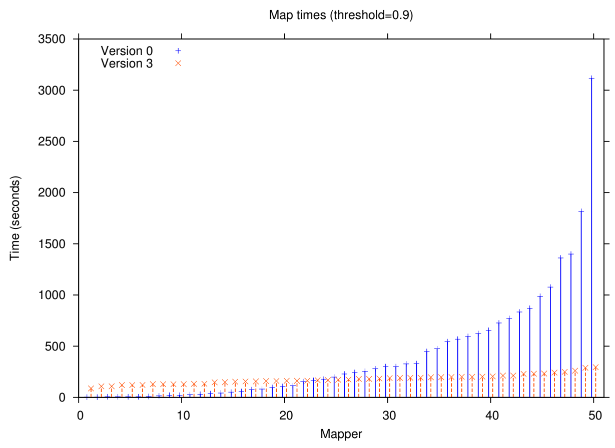

Figure 5: Map time distribution with and without bucketing

of map times for versions 1, 2 and 3 is much lower than version 0 thanks to bucketing.

For average reduce time, things change with different input sizes. For small inputs the overhead of version 3 does not pay back, and version 2 has the best trade-off between algorithm complexity and number of partial scores. For large inputs the smaller number of partial scores of version 3 gives it an edge over other versions. Version 1 is the slowest because of disk access and version 0 also scales poorly because of the large number of partial scores.

5. CONCLUSIONS AND FUTURE WORK

The All Pairs Similarity Search problem is a challenging problem that arises in many applications in the area of information retrieval, such as recommender systems and near duplicate detection. The size of Web-related problems mandates the use of parallel approaches in order to achieve reasonable computing times. In this

We presented a novel exact algorithm for the APS problem. The algorithm is based on the inverted index approach and is developed within the MapReduce framework. To the best of our knowledge, this is the first work to exploit well known pruning techniques from the literature adapting them to the MapReduce framework. We evaluated several heuristics aimed at reducing the cost of the algorithm. Our proposed approach runs up to 5x faster than the simple algorithm based on inverted index.

In our work we focused on scalability with respect to the input size. We believe that the scalability of the algorithm with respect to parallelism level deserves further investigation. In addition, we believe that more aggressive pruning techniques can be embedded in the algorithms. Adapting these techniques to a parallel environment such as MapReduce requires further study. We also want to investigate the application of our algorithm to other kinds of real world data, like social networks.

6. ACKNOWLEDGEMENTS

This work was partially supported both by the EU-FP7-215483 (S-Cube) and the POR-FESR 2007-2013 (VISITO Tuscany) projects.

References

[1] Apache Software Foundation. Hadoop: A framework for running applications on large clusters built of commodity hardware, 2006.

[2] A. Awekar and N. F. Samatova. Fast Matching for All Pairs Similarity Search. In *WI-IAT '09: Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, pages 295–300. IEEE Computer Society, 2009.

[3] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 131–140. ACM, 2007.

[4] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI '04: Proceedings of the 6th Symposium on Opearting Systems Design and Implementation*, pages 137–150. USENIX Association, December 2004.

[5] T. Elsayed, J. Lin, and D. W. Oard. Pairwise document similarity in large collections with MapReduce. In *HLT '08: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies*, pages 265–268. Association for Computational Linguistics, 2008.