DISTRIBUTED SYSTEM FOR DISCOVERING SIMILAR DOCUMENTS

From a Relational Database to the Custom-Developed Parallel Solution

Jan Kasprzak, Michal Brandejs, Miroslav Křipač and Pavel Šmerk
Faculty of Informatics, Masaryk University, Botanická 68a, Brno, Czech Republic

Keywords: University, Plagiarism, Similar Documents, Cluster, Information System, Theses.

Abstract: One of the drawbacks of e-learning methods such as Web-based submission and evaluation of students' papers and essays is that it has become easier for students to plagiarize the work of other people. In this paper we present a computer-based system for discovering similar documents, which has been in use at Masaryk University in Brno since August 2006, and which will also be used in the forthcoming Czech national archive of graduate theses. We also focus on practical aspects of this system: achieving near real-time response to newly imported documents, and computational feasibility of handling large sets of documents on commodity hardware. We also show the possibilities and problems with parallelization of this system for running on a distributed cluster of computers.

1 INTRODUCTION

1.1 About IS MU

At Masaryk University, the study administration is being supported by a web-based Information System (`http://is.muni.cz/`, IS MU), which has been in development since 1999. See (Pazdziora and Brandejs, 2000) for detils about IS MU. Since then, IS MU has become the central part of the study administration and communication at Masaryk University. Among others, it handles e-learning tasks like submitting essays, and it stores various documents such as students' theses. The in-house developed distributed storage subsystem is used for these tasks.

1.2 Handling Plagiarism

One of the problems of storing (and making available) documents in an electronic form is that documents can be easily plagiarized. This is by no means a problem specific to IS MU: students often have their own WWW sites for exchanging documents like essays or written exams, so disallowing document sharing inside IS MU would not help to mitigate the problem.

Instead, we actively encourage document sharing, and using old essays as basis for new ones, *provided that the source is correctly cited*. However, we must provide tools to detect similar documents, so that the teacher (or a thesis reviewer) can easily discover copied sections in students' essays. The actual decision whether the document is plagiarized or not relies on human work. The machine can only serve as a tool.

In this paper, we will discuss the inner workings of our system for discovering similar documnets in its original prototype SQL database-backed form (which has been in use inside IS MU since August 2006), and in its new implementation, which will be more than an order of magnitude faster, while allowing it to be distributed to a set of commodity computers in a Linux cluster. This system will be used also in the forthcoming Czech national archive of graduate theses.

2 SIMILAR DOCUMENTS

Firstly let us describe which documents we consider similar and how to calculate similarity in documents. There are various approaches in discovering similar documents (Monostori et al., 2002). We use a chunk-

Kasprzak J., Brandejs M., Kripač M. and Šmerk P. (2008).
DISTRIBUTED SYSTEM FOR DISCOVERING SIMILAR DOCUMENTS.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - DISI*, pages 437-440
DOI: 10.5220/0001687604370440

based approach: the document in its plain text form is split into chunks of text, and the system then tries to find these chunks inside other documents.

2.1 Document Similarity

For two documents A and B we define the *similarity of the document A to the document B* as a percentage of chunks of the document A which can also be found inside the document B. Using this definition, the similarity is a real number between 0 and 100 inclusively.

Note that similarity is not symmetric (for example, when the document A as a whole is contained inside a bigger document B). The actual similarity of the document B to the document A can be computed as

$$\frac{\text{number of common chunks in } A \text{ and } B}{\text{total number of chunks in } B} \cdot 100\%$$

2.2 Current Data Set

We have approximately 250,000 documents in IS MU which enter the similarity search process. This set of documents can be transformed to about 600,000,000 (chunk, document-ID) pairs. There are circa 445,000,000 unique chunks in the data set. For the Czech national archive of graduate theses, it is expected that the total volume will be at least two times bigger.

3 PROTOTYPE SYSTEM

In the first implementation, we have used Oracle as the database back-end. We have not stored the chunks themselves, but a concatenation of word ID numbers from the dictionary instead, which has saved us some space.

3.1 Resource Requirements

The data needed for calculating the similar documents has been stored in the following database tables:

dictionary—for converting betwen the word and its ID. The table had 900,000 rows in 19 MB of disk space, and both indexes in another 36 MB.

chunk table—mapping the document ID to the chunk. The table data has about 30 GB, the index which maps the chunk to the document ID has about 46 GB, and the reverse index 17 GB.

The system runs on a SGI Altix 350 with 14 Itanium2 CPUs and 28 GB of RAM. The big amount of RAM is significant, but the data set is still bigger than the available memory.

Generating chunks from all the documents takes about two hours, inserting them to the chunk table takes another two hours (both using all 14 CPUs). Computing similarities from the chunk table takes about 50 hours also using all 14 CPUs.

3.2 Pros and Cons

Interestingly enough, the Oracle representation of the chunk table was not much bigger than expected—their metadata size did not add any substantial overhead. To obtain a significant speed improvement, the different data structures will have to be used. Also the solution with SQL database and Perl/DBI can be easily prototyped, so we could put the system into the production use relatively fast.

On the other hand, the ACID properties of SQL database have been a bottleneck of the system. For most of the tasks we did not need a strictly isolated transactions.

4 DISTRIBUTED APPROACH

In the next step, we have decided to reimplement this system outside the database, in the tightly packed and customized data structures. The requirements to the new system were the following:

- Usability on a commodity hardware with much less resources than our Altix system.
- Scalability by by adding computing nodes, not expanding the single server.
- Speed. Users are not willing to tolerate several hours or even a day of delay for newly added documents.

4.1 Chunk Table

The biggest barrier which prevents the system from being used on commodity hardware is the size of the chunk table. Even our mid-range server cannot fit the data set into its memory. The estimated size limits of this approach are:

We have about 1 milion words in the dictionary. So we need about 20 bits to encode a word. With average five-word chunks, we need 100 bits, i.e. 12.5 bytes, to encode the chunk itself. With 450,000,000 different chunks, we would need about 5.2 GB to store just the chunk IDs in this extremely tightly packed encoding, not counting the documents in which those chunks appear, and the index needed

for fast searching inside this data. Thus We need to shrink the data in the chunk table even more.

4.2 Chunk as its Hash

We propose that the chunk identification should be stored not as the exact set of word identifications, but as some kind of the hash value of the words themselves. This gives us a lower number of bits needed for expressing the chunk identification. Moreover, by using different hash functions we can even choose the number of bits used for expressing the chunk ID. In other words, we can set the various levels of tradeoff between the data size and the accuracy of the data (the probability of hash collisions).

The hash function does not matter, we can for example take the highest n bits of MD5(chunk). As for the value of n, we have tried values of 24 and 28 bits. Note that the total number of different chunks in our data set is between 2^{28} and 2^{29}. The results were interesting: with 24 bits of hash value size, the absolute difference between the computed and exact similarities were up to 5 %, but only for documents which had their similarity already at most 5 %. So we have got only few false positives for the document pairs which have already been different enough. For $n = 28$, the absolute difference was at most 1 %.

Should the exact results be needed, we can use this approach as an upper estimate of the similarity, and compute the exact similarities only for document pairs which are preselected by this algorithm, and only after the user looks at these documents (so not precompute the exact values).

Also note that using hash from the words themselves relaxes the need of unique word ID numbers. The dictionary table then can be transformed into a set (i.e. we will not have to look up the word ID, but instead only ask whether the word is present in the dictionary or not). This can lower the resource requirements for the dictionary table, altough this reduction is not significant in the whole picture.

4.3 Data Structure

The hash function we use has the range of values from 0 to $2^n - 1$ for some n. Unlike the database approach, we actually do not need the whole chunk table, searchable both by chunk ID and the document ID. In fact, we only need one of these two directions: for discovering similar documents to a given one, we need to split the new document into the chunks, and then look up in which other documents those chunks are. So in the database speech, we only need the *index* mapping the chunk ID to the list of document IDs. The proposed data structure for this task is described in the Figure 1. The data structure contains two arrays:

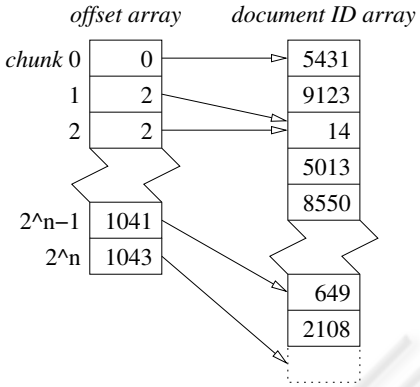

Figure 1: Data structure mapping chunk ID to the document IDs.

- The *array of document IDs* (in the Figure 1 the rightmost one). This is an array of values of the "document ID" data type. It contains the list of documents in which the ID 0 appears, then the list of documents in which the chunk ID 1 appears, and so on. The size of this array is approximately equal to `sizeof(document_id)` multiplied by the total number of all chunks in all documents. For 600,000,000 chunks and three bytes for the document ID, it is about 1.7 GB. There is nothing simple we can do to reduce the size of this array.
- The *array of offsets* (in the Figure 1 the leftmost one). This array describes where in the array of document IDs we should look, when we want to find all documents, in which a given chunk occurs. The entry i of this array gives the offset of the first document ID for the chunk with the hash value i, and the entry $i+1$ gives the offset of the first document ID, in which this chunk *does not* occur. It is an array of the integer data type, indexed by all possible values of the chunk ID. So for 24-bit hash function value space and 4-byte integer, this array has $2^{24} \cdot 4$ bytes, i.e. 64 MB, and for 28-bit hash function value space it has 1 GB. So the size of this array is proportional to the number of bits of the hash value.

For example, in the Figure 1, the chunk with hash value of 0 occurs in documents 5431 and 9123, the chunk with hash value 1 is not anywhere in the whole data set, the chunk with hash value 2 is in documents 14, 5013, 8550 and maybe others. The 2^n-th entry is used to terminate the array of document IDs.

4.4 Algorithm

1. Construct a set of hash-based chunk IDs of all not yet added (i.e. new) documents.
2. Construct the array of document IDs and the array of offsets as described in Section 4.3.
3. Merge the data structure from the previous step with the same data structure describing the previously added documents, possibly removing data about documents, which has been deleted from the system.
4. Using the merged data structure, for each newly added document find all documents similar to it. If similarities are found in the documents which already had been in the database from previous runs of this algorithm, also compute the inverse similarity (as described in Section 2.1).

4.5 Properties of the Algorithm

- As for transforming the plain text form of the document to the set of chunks, there is not much to be improved speed-wise. This is an easily parallelized task, and id does not need any network communication (other than retrieving the document itself and storing the computed results).
- In the step 2 we want to compute an "inverted index". I.e. from the document ID to list of chunk IDs mapping, we need to compute the opposite direction. We can use a bucket sort especially as the data can be pre-sorted into a given number of buckets in the step 1.
- Merging the two data structures from Section 4.3 can be done sequentially, in a linear time. This step cannot be parallelized. However, we can split the whole data structure to the cluster nodes giving each node its own range of the chunk IDs. Then each node can merge only its own part of the data structure.
- Finding similar documents: the complexity of this step is proportional to the number of chunks in the newly added documents. We can distribute this task so that each cluster node handles only part of the document ID range. So by adding more nodes, we lower the memory requirements on each node.
- Incremental runs: the incremental runs are fast, we expect them to be run in a one- to five-minute period on a production system.

4.6 Practical Results

We have implemented this algorithm, and we are able to present some practical results:

- The step 1 took about 2 hours, including presorting different chunk ranges to separate files, in order to do a radix sort in the next step. The time taken is about the same as in the prototype solution.
- The step 2, i.e. merging the pre-sorted ranges, took about three hours on a single CPU. Further speed improvements by using multiple nodes or multiple CPUs are possible by, for example, giving each node its own range of chunk IDs to sort.
- The resulting data structure takes less than 2 GB of memory for 24-bit hash value, and less than 3 GB of memory for 28-bit hash value.
- Finding similar documents using this data structure can be done in slightly over two hours on 14 CPUs.

Thus the total run time of this new system for the inital recomputing all similarities in the given data set is about 7 hours. Preliminary results with the fully distributed implementation on a cluster of 22 dual-core nodes shows that the total run time should fit into one hour.

5 CONCLUSIONS

We have described two generations of a system for finding similar documents in the real-world information system.

The new implementation runs much faster than the prototype one (7 hours versus 54 hours for an initial step), with more speedup possible. No part of the new system require more than 4 GB of RAM, and it can be distributed on a cluster of commodity computers.

So far we are not aware of any other system for finding similarities in documents, which uses the hash-based approach for approximating the actual chunk identification. This approach can provide significant savings in the total memory needed.

REFERENCES

Monostori, K., Finkel, R. A., Zaslavsky, A. B., Hodász, G., and Pataki, M. (2002). Comparison of overlap detection techniques. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part I*, pages 51–60, London, UK. Springer-Verlag.

Pazdziora, J. and Brandejs, M. (2000). University information system fully based on www. In *ICEIS 2000 Proceedings*, pages 467–471. Escola Superior de Tecnologia do Instituto Politcnico de Setbal.