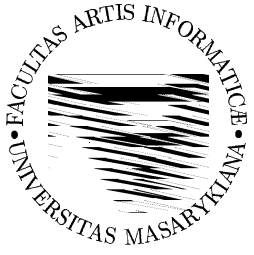

FI MU

Faculty of Informatics
Masaryk University Brno

Distributed System for Discovering Similar Documents

From a Relational Database to the Custom-Developed Parallel Solution

by

Jan Kasprzak
Michal Brandejs
Miroslav Křipač
Pavel Šmerk

FI MU Report Series **FIMU-RS-2008-04**

Copyright © 2008, FI MU **June 2008**

**Copyright © 2008, Faculty of Informatics, Masaryk University.
All rights reserved.**

**Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.**

Publications in the FI MU Report Series are in general accessible via WWW:

`http://www.fi.muni.cz/reports/`

Further information can be obtained by contacting:

**Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic**

Distributed System for Discovering Similar Documents

From a Relational Database to the Custom-Developed Parallel Solution

Jan Kasprzak Michal Brandejs Miroslav Křipač
Pavel Šmerk
Faculty of Informatics, Masaryk University
Brno, Czech Republic
{kas,brandejs,kripac,smerk}@fi.muni.cz

June 30, 2008

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]

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). The system has been in development

[1]Short version of this paper has been presented at the 10th International Conference on Enterprise Information Systems (`http://www.iceis.org/`), and is available in the conference proceedings.

since 1999. For basic premises on which the IS MU is being developed, see the early paper from ICEIS 2000 [2]. Since then, IS MU has become the central part of the study administration and communication at Masaryk University.

As a background information, we provide some traffic statistics: IS MU is being actively used by about about 27,000 different users each day (from the total of about 140,000, who have an active account in IS MU). These system renders up to 2,000,000 authenticated dynamic WWW pages for them daily. The system is also provided in an outsourcing form to some other universities in the Czech Republic.

1.2 E-learning Subsystem

Of the subsystems which have been developed or improved inside IS MU in the last few years probably the most actively developed (and also the most wanted by users) are those related to e-learning. IS MU supports storing study materials, including the system of access rights, discussion forums (course-wide ones as well as global ones), WWW-based e-mail server, electronic test forms, automatic recognition and evaluation of paper question forms, collecting students' essays and tools for evaluation of these, and so on.

1.3 Document Storage

The study materials from the e-learning subsystem are stored in an in-house developed distributed and replicated data storage, running on a cluster of Linux servers, equipped with commodity hard disks. This storage system allows cheap and reliable storage of large amounts of data. The storage system is used also by other agendas like user e-mail boxes, public university documents, and last but not least, the graduate theses.

The document storage provides many features—some of them similar to the filesystem features (like a tree-organized structure, file names, hard links for study materials of the same course in different semesters, etc.), and some which are related to the data stored: automatic background conversion of files in proprietary formats like Word or Excel to open formats like PDF and plain text, character set encoding detection, image thumb-nails, file descriptions, automatic or manual setting of MIME types, time-related access rights (used, for example, for automatically revoking the "create file" permission in folders dedicated for uploading students' homeworks after the deadline), etc.

1.4 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 submitted document is plagiarized or simply contains some quoted text, which is correctly labeled as such, relies on human work. The machine can only serve as a tool.

1.5 Thesis Archive

The IS MU development team has started working on a next project, the Czech national archive of graduate theses. This project will use the same distributed document storage as IS MU. Because this project will make a huge number of theses available on-line, we will also have to provide tools for discovering similar works in this document base.

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.

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 [1]. We use a chunk-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.

Please note that similarity is not symmetric: for example, when the document A as a whole is contained inside a bigger document B (think thesis template with the license agreement and so on, and the thesis based on this template), the document A is 100 % similar to the document B, while the the document B similarity to the document A is lower than 100 %. However, because the absolute number of common chunks in the documents A and B is the same, the actual similarity of the document B to the document A can be computed as

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

2.2 What is a Chunk

One of the tricky parts of this problem is how to construct the set of chunks for a given document. We construct chunks as several consecutive words from the document (so we skip any non-word characters like interpunction). For our purposes, five-word chunks are sufficient. The chunks are constructed from each five consecutive words in the documents, so they are overlapping.

Additionaly, we sort the words inside each chunk. This at the first sight may look like we are lowering the algorithm precision, but it is not the case: sorting the words in chunks can help to overcome common tricks like word transposition. Czech is more-or-less a free word order language, where *some* word transpositions can still lead into a fully legible text.

As an example, the first four sorted five-word chunks constructed from the previous paragraph would be the following:

1. `additionaly sort the we words`
2. `inside sort the we words`
3. `each inside sort the words`
4. `chunk each inside the words`

2.3 Current Data Set

We have approximately 250,000 documents in IS MU, which have its plain text form, and which have not been excluded from the process of finding similar documents (an example of such excluded documents are discussion forum posts, documents in users' temporary file repositories, etc.). This set of documents can be transformed to about 600,000,000 (chunk, document-ID) pairs. There is 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 quickly increase (the initial rough estimate is that the total volume will be twice as big as the current data set in the first year of usage), while the number of documents will probably not increase significantly: the current data set in IS MU contains not only relatively long theses, but also much shorter essays, seminar works, articles, etc.

3 Prototype System

The first implementation of the above approach has been in the production use in IS MU since the August 2006. We have used Oracle as the database back-end.

We have not stored the chunks themselves, but a sorted concatenation of word ID numbers from the dictionary instead, which has saved us some space.

3.1 Data Structures

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

dictionary—a table of two columns: word ID and the word itself. We use dictionaries for Czech, Slovak, and English languages, but it is easy to add more. The table has slightly under 900,000 rows, which occupy about 19 MB of data space. The table has indexes on both word ID and the word itself, these indexes take additional about 16 MB and 20 MB, respectively.

chunk table—a table of six columns: document ID, and IDs of the five words in 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 for looking up all chunks corresponding to a given document ID has slightly under 17 GB. The first index

is needed for looking up similar documents, and the second one is needed for removing entries which belong to the documents, which have already been deleted from the documents repository.

similarity database—this table contains the computed results, i.e. IDs of two documents, and their calculated similarity. It contains about 8 milion rows, while we insert into this table only pairs of documents with the similarity of at least 1 %.

The above table and index sizes are raw data segment capacity from the live system, as reported by Oracle. The actual data size is a bit lower.

3.2 System Performance

The system runs on our former main database server, 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.

The system then works in the two steps:

- Firstly the existing set of documents as a whole is recalculated: documents are transformed into set of chunks in the database table, and then similar documents inside the whole set are computed from the information in the chunk table.
- The next step is being run on a regular basis—for any newly uploaded document the same has been done: transforming it into chunks, and finding whether these chunks are also included in the existing set of documents, and computing the similarity of this documents to the existing documents.

In the first step generating chunks from the documents takes about two hours, inserting them to the chunk table in the database takes another up to two hours (both using all 14 CPUs). Computing similarities from the chunk table takes about 50 hours also using all 14 CPUs.

The second step was not without problems: when a bigger number of documents has been uploaded to the system (such as mass import of scanned theses from the archive), the recalculation of the similarity took up to one day. So the similarity interface in the WWW system cannot be used by for example teachers who wanted to know quickly whether some of just-submitted seminar works is plagiarized or not.

3.3 Pros and Cons

The original system has showed us some interesting facts: firstly, 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 as a backing store and Perl with the DBI interface can be easily prototyped, so we could put the system into the production use relatively fast.

On the other hand, the strict ACID properties of SQL database have been a bottleneck of the system. For most of the tasks (such as the first step of the above algorithm), we did not need a strictly isolated and atomic transactions—the data will be usable only after all the documents are transformed into the chunks anyway.

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:

- The system should be usable even on a commodity hardware with much less resources than our Altix system.
- The system should be made scalable not by adding CPU and memory to the server, but by adding another computing node. The commodity hardware has relatively low upper limits on the amount of supported memory. On the other hand, adding another node to the cluster is cheaper, and more importantly, almost always possible.
- The new system should be faster. Users are willing to tolerate few minutes delay between the time the new document is inserted and the time the system can find the documents similar to it. However, they are not willing to tolerate several hours or even a day of delay.

4.1 Chunk Table

The biggest barrier which prevents the system from being used on commodity hardware is probably the size of the chunk table. With the approach described in the previous section, even our mid-range server cannot fit the data set into its memory. Let's estimate the theoretical size limits of this approach:

We have about 1 milion words in the dictionary. So we need about 20 bits to encode a word. With 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.

We need to shrink the data in the chunk table even more.

4.2 Chunk as Its Hash

In order to reduce the memory space needed, 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).

We have chosen a function which is has random enough distribution, the MD5 digest function. Until recently, it has even been considered cryptographically strongi so the distribution is sufficiently random, yet MD5 is a bit hard to compute. To obtain the desired number of bits of the hash value, we have just took the upper n bits of $\mathrm{MD5}(\mathrm{word}_1 \cdot \mathrm{word}_2 \cdot \mathrm{word}_3 \cdot \mathrm{word}_4 \cdot \mathrm{word}_5)$.

As for the value of n, we have tried values of 24 and 28 bits. Note that the total number of different chunks in the given data set is between 2^{28} and 2^{29}. The results were interesting: with 24 bits of hash value size, the difference between the computed and expected (exact) results were up to 5 %, but only for documents which had their similarity already at most 5 %. So we have got only some false positives for the document

pairs which have already been different enough. For $n = 28$, the computed value was not different by more than 1 % from the expected value[2].

Should the exact results be needed, we can always use this algorithm 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 ID.

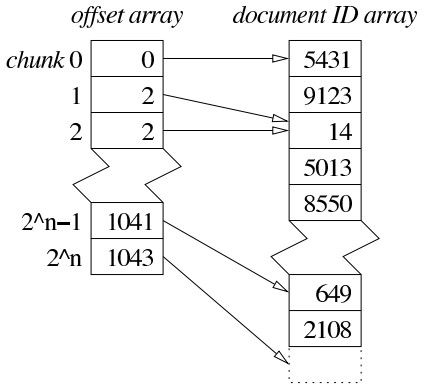

Figure 1: The data structure mapping chunk ID to the list of document IDs.

[2]Note that the similarity is defined in Section 2.1 as a percentage. So the above "not more than 1 %" means that when–for example–the computed similarity of the document A to the document B was 2 %, the expected one was between 1 % and 3 %, inclusively.

The proposed data structure for this task is described in the Figure 1. The data structure contains two arrays:

- 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. This means the bigger the hash value range is, the more memory we will need, but the lower probability of hash collisions will be, and the results given by our algorithm will be more accurate.

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, and the last chunk (hash value $2^n - 1$) occurs in the documents 649 and 2108. Note how the entry 2^n in the array of offsets is used to determine the number of entries for the hash value $2^n - 1$ in the array of document IDs.

For further possible optimizations we keep the sub-list of document IDs for a given chunk sorted.

4.4 Algorithm

The actual task of finding similar documents in a given set of documents runs in the following steps:

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 (from the number of common chunks and from the total number of chunks in this older document as described in Section 2.1).

Note how we do not have a separate "compute everything" step, and it is just a special case of the "new documents added" step, starting with an empty array of document IDs and an empty array of offsets. However, for the initial recomputing of the whole data set it may not be feasible to keep the list of chunks for a given document ID computed in the step 1, and instead we can recompute it again as-needed in the step 4.

4.5 Properties of the Algorithm

Let us discuss the computing resources needed for the above 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 provided that the dictionary is replicated to the cluster nodes, it even 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, mapping the chunk ID to the list of document IDs. This can be done for example by a variant of a radix sort (especially when the whole data set does not fit into the memory, such as in the "compute everything" phase). The radix sort can be further speeded up by pre-sorting the data 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. However, this step uses both big arrays only for reading, so this step can be fully parallelized (on a shared-memory machine by mapping the arrays to multiple processes or threads, and on a cluster by copying the whole arrays to each cluster node, which can be done in sub-minute times). Should the size of these two arrays grow outside the available RAM, 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 pre-sorting 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 runs on one CPU at speeds around 8500 documents per hour, so the whole data set can be recomputed in slightly more than 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, while still having room for improvement by parallelization.

5 Future Work

There is a number of areas where further improvements can be made:

- *Generating chunks:* so far we have got along without using lemmatization (i.e. finding a basic form from the inflected word). We think that for for inflectional languages like Czech or Slovak, it can make a improvement in recognizing similar parts of the text (in Czech, for example, when swapping two consecutive words, usually a different inflection has to be used).
- *Different hash function:* we can do some experiments with hash functions other than based on MD5. MD5 is sufficiently random, but relatively slow to compute (after all, it has been designed to be cryptographically strong). However, this part of the algorithm can be easily parallelized, so this probably will not result in a meaningful improvement of the whole process.
- *Better encoding of the array of offsets from the Section 4.3:* since most chunks appear in very low number of documents (most often there are in 0 or 1 document), so the offset table can be compressed: the neighbouring values are usually close to each other, so some kind of differential encoding could save more memory space as a trade-off for more CPU time.

6 Conclusion

We have described two generations of a system for finding similar documents in the real-world information system. One of the most important properties of this system is that it is not only a theoretical construct, but a real-world system, which has been used since the August 2006. The strength of this system both in its abilities, but also (as we have to admit) in its pure existence: students are now aware that it is easy for the teachers of our University to find similar documents, so they are less motivated to plagiarize work of other people.

The new implementation runs much faster than the prototype one (7 hours versus 54 hours for an initial step). And there is still room for further improvements. No part of the new system require more than 4 GB of RAM, while the system can be run on a cluster of nodes with even less RAM available. The new implementation is usable both on clusters and shared-memory multiprocessors.

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. As we have verified, this approach can provide significant savings in the total memory needed, allowing the system to be run on a commodity hardware, while the accuracy of the computed values remain within a reasonable distance from the exact values.

References

[1] Krisztián Monostori, Raphael A. Finkel, Arkady B. Zaslavsky, Gábor Hodász, and Máté Pataki. Comparison of overlap detection techniques. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part I*, pages 51–60, London, UK, 2002. Springer-Verlag.

[2] J. Pazdziora and M. Brandejs. University information system fully based on www. In *ICEIS 2000 Proceedings*, pages 467–471. Escola Superior de Tecnologia do Instituto Politécnico de Setúbal, 2000.