DISTRIBUTED ASPECTS OF THE SYSTEM FOR DISCOVERING SIMILAR DOCUMENTS

Jan Kasprzak[1], Michal Brandejs[2], Jitka Brandejsová[3]

[1]Faculty of Informatics, Masaryk University, Czech Republic
kas@fi.muni.cz
[2]Faculty of Informatics, Masaryk University, Czech Republic
brandejs@fi.muni.cz
[3]Faculty of Informatics, Masaryk University, Czech Reublic
brandejsova@fi.muni.cz

ABSTRACT

With wide deployment of e-learning methods such as computer-mediated communication between the students and teachers, including papers and essays submission and evaluation, it has become much easier for students to base those works on electronic resources, including the plagiarization of the work of other people. In this paper we will briefly present a system for discovering similarities in a large base of documents, which has been in production use inside the Czech National Archive of Graduate Theses since January 2008. We will then focus on the distributed aspects of such a system, especially on the task of creating and maintaining the index for discovering the similarities on a cluster of commodity computers.

KEYWORDS

Theses, Archive, Plagiarism, Similar documents, Distributed computing

1. INTRODUCTION

1.1. The Theses Archive

Since 2006, the universities in the Czech Republic are required by law to make the graduate theses of their students publicly available. Having the results of publicly funded education and research available for the general public means that the quality of the education can be closely scrutinized by any interested party.

Some universities have decided that the most transparent way of fulfilling the law is to make the texts of the theses available on the web. In late 2007, the effort of several universities and university libraries resulted in a project of the Czech National Archive of Graduate Theses [1], sponsored by the Czech Ministry of Education. The system has been developed and deployed in 2008 by the team from the Masaryk University in Brno. As of February 2009, there are about 20 universities participating in the theses.cz system.

Having a central archive of graduate theses definitely lowers the barrier for accessing these texts (which would otherwise require to physically visit or at least communicate with the library of some remote faculty or university). On the other hand, having the full texts of theses available for downloading and reading also means that it is easier to copy the work from other texts. Thus a necessary component of the theses archive is the system for discovering similarities in a given document base.

1.2. University Information System

The theses.cz archive shares a big part of code base with the Masaryk University Information System (IS MU, [2], [3]). In the e-learning subsystem of IS MU, there are tools for submitting students' essays, evaluating them, and also searching and finding similarities in them. The system for finding similar documents in IS MU has been deployed since 2006. For the theses.cz archive, it has undergone a major rewrite, which allows faster response to newly added documents, as well as distributed processing of the documents.

1.3. Document Storage

The crucial part of both IS MU e-learning subsystem and the theses.cz archive is the document storage subsystem. It is an in-house developed networked data storage, running on a cluster of commodity computers with the Linux operating system, storing data on cheap commodity hard disks. The storage subsystem provides some features similar to the features of general-purpose file systems (such as tree-organized structure, file names, hard links, etc.).

Apart from that, it also has some features unique to it, like alternative versions of documents, most of which are created automatically (e.g. when user imports a Word or OpenOffice.org document, it is automatically converted also to PDF, and the plain text is extracted from it; for the imported PDF files, the plain text extraction can use the PDF file properties, or - should the PDF file be bitmap-based - the text can be extracted by the means of the OCR software). Having a reliable plain text version of all documents is a necessary prerequisite for being able to search these documents, and also being able to find similarities in the document base.

Amongst other special features we can mention replicating, periodically verified checksums, or rich system of access rights (such as "students of such and such course in this semester" or "student which had enrolled in such and such course in any past or present semester", etc.).

The document storage is used not only for the e-learning agendas in IS MU, but also for other purposes like the back-end for user e-mail boxes, or a bulletin board of the university documents. As of February 2009, the storage subsystem of the IS MU and theses.cz has a raw storage capacity of about 80 terabytes, and currently hosts about 13 millions of objects, amongst them about 1,200,000 documents which are indexed for the purpose of finding the document similarities (theses, seminar works, articles, etc.).

2. NON-DISTRIBUTED ALGORITHM

2.1 Document similarity

For finding the similarities in the document base, we use a chunk-based approach to the finding similarities problem: roughly speaking, the plain text form of the document is split into short, overlapping sequences of words (chunks), and we look up those sequences in other documents. The similarity of the document A to the document B is defined as a number of chunks from the document A, which are also present in the document B (note that the similarity is not symmetric - e.g. a shorter document A can be included as a whole in the larger document B - then A is 100 % similar to the document B, while the document B can be only a few percent similar to the document A).

2.2 Non-distributed algorithm

The non-distributed version of the algorithm we use has been described in detail in our earlier work [4], including the performance analysis on a real-world set of documents. The algorithm works the following way:

1. For each document, do
    a. generate the chunks from the plain text form of a document
    b. from the chunk generate its ID using a hash function
    c. store the (*document ID, list of chunk IDs for this document*)
    d. store the number of chunks in this document
    e. generate a list of (*this document ID, chunk ID*) pairs
    f. split the chunk ID space to several parts (buckets), add the (*this document ID, chunk ID*) pairs into appropriate buckets
2. For each bucket generated in step 1f, do
    a. sort the bucket by the chunk ID
    b. generate the (*chunk ID, list of document IDs*) data structure
    c. add this data structure to the data structures computed earlier for previous parts of the chunk ID space
3. For each document *d1*, do
    a. create an empty list of pairs (*document ID, number of shared chunk IDs*)
    b. take the list of chunk IDs for this document (as computed in step 1c),
    for each chunk ID in this document *d1*, do
        i. in the data structure constructed in step 2c, look up the list of documents which also contain this chunk ID.
        ii. for each of those documents (except *d1*), add an entry to the list created in step 3a, or if the entry for a given document ID is already present, increment the counter of shared chunk IDs.
    c. have a table of precomputed document similarities: triplets of (*document 1 ID, document 2 ID, similarity*)
    d. for each entry (*document ID d2, number of shared chunks n*) in the list generated in steps 3a and 3b-ii, do
        i. compute the similarity *s* as *n*/(number of chunks in the document *d1*, as computed in step 1d)
        ii. add (*d1, d2, s*) to the table from step 3c.

We have verified that even though we use hash-based chunk IDs which are not always unique because of the hash function collisions, the results are not affected in a significant way (we have observed a variance within one percent of the exact value).

Using a fixed width chunk ID (we have tested 28, 30, and 32 bits) allows us to split the chunk ID space in the step 1f in preparation for a bucket sort.

Also note that in steps 1f and 2a-c, we essentially do a bucket sort, which computes an inverted index mapping a chunk ID to the list of document IDs from the original mapping of a document ID to the list of chunk IDs.

2.3. Data structures

The resulting invented index is kept in two separate files/arrays:

- *array of document IDs* (the rightmost one in Figure 1) is an array of the document ID data type, and it contains the sorted list of IDs of documents which contain the chunk with ID 0, then the list of IDs of documents which contain the chunk ID 1, then the same for chunk ID 2, etc.
- *array of offsets, mapping the chunk ID to the appropriate part of the array of document IDs*.

For example, in Figure 1, the chunk with ID 0 is present in documents number 243 and 5039, the chunk with ID 1 is not present anywhere in the base of documents, and the chunk with ID 2^n-1 is present in documents 649 and 2108.

The size of the array of document IDs is equal to the number of different (chunk ID, document ID) pairs in the whole base of documents (for our document base, it has about 8.9 GB with 32-bit document IDs).

The size of the array of offsets is (2^n+1) * *size of the pointer*. For 32-bit pointers and n=30, we have 4 GB plus 4 bytes.

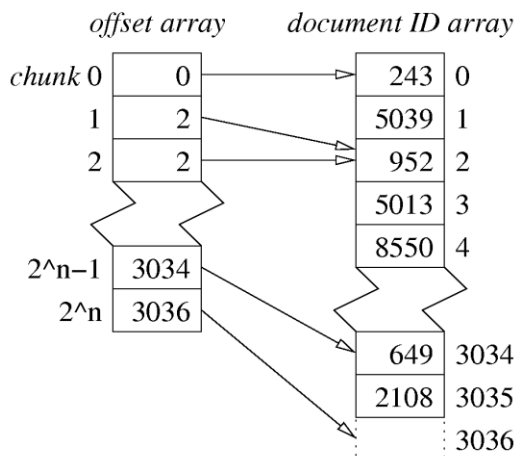

Figure 1: Data structure for the inverted index

This data structure needs to be stored permanently for incremental runs of the algorithm, when only a few documents are added/modified/deleted.

The list of document similarities as needed in steps 3a, 3b-ii, and 3d can be kept in memory. Any tree-based data structure can be sufficient, we use a Judy array library as described in [5].

2.4. Incremental runs

One of the task the system has to fulfil is to have a fast response time to a newly imported documents. Users who import their documents should be able to search for similar documents to the just-imported documents soon after the import is done. It is not feasible to run the whole Algorithm 1 periodically. We have designed the data structures to make an incremental variant of Algorithm 1 possible. Here are the modifications required for incremental runs:

1. Before the step 1, we have to find out which documents were added, deleted, or modified since the last run of the algorithm.
2. The steps 1 and 3 are run only for documents, which have been added or modified.
3. After the step 2, we have to merge the inverted index we have just computed with the previous one. We walk through the original inverted index, and delete the document IDs which were deleted or modified since the last run from the array of document IDs, and adjust the array of offsets appropriately. We can merge the result with the newly computed inverted index after that, or even while we are still walking the original inverted index.
4. Before the step 3, we have to delete the entries (*document 1 ID, document 2 ID, similarity*) from the table of similarities for all rows where *document 1 ID* or *document 2 ID* is amongst the documents which has been modified or deleted since the last run.
5. In step 3d-ii when the *document 2 ID* is not amongst the documents added or modified since the last run, compute also the reverse similarity of document 2 to the document 1 as number of common chunk IDs divided by the number of chunks in document 2. Note that

we do not need to do that for newly added or modified document 2, because the other direction is also recomputed in the step 3d-ii.

For the step 1, we need an in-memory data structure which can be easily searched for the presence of the key. Again, we use the 1-bit Judy array for this.

For the incremental version of this algorithm, we in addition to the inverted index (array of offsets, and array of document IDs) need to save the mapping of the document ID to the number of chunk IDs in this document, which is needed in step 5 of the incremental run.

In order to keep a document ID range low, we recycle the IDs of the previously deleted documents, instead of having forever increasing IDs taken from, for example, a database server sequence object.

3. DISTRIBUTED ALGORITHM

The algorithm and the data structure described in the previous section were designed in order to be easily parallelized to a cluster of computers. In short, we can easily split the chunk ID space equally between the computing nodes.

The method of computing an inverted index on a cluster of computers has been described in the paper on the MapReduce system [7]. We use a similar approach, but in addition to the inverted index, we have to perform additional computations with the newly computed index (step 3 of the non-distributed algorithm).

3.1. Algorithm

1. Determine the number of nodes participating in this run, and number of processes running on each node. Start those processes, communicating with the master process.
2. Split the chunk ID space equally between the processes, inform each client process about the configuration of the computing network and about the parts of chunk ID space assigned to particular processes. Each client process now makes a network connection with each other client process.
3. From the master process, split the previously computed inverted index (i.e. array of document IDs and array of offsets) according to the chunk ID space assignment, and send the part of this data structure to each respective client process.
4. From the master process, determine which documents have been deleted, added or modified since the last run. For each added, modified or deleted document, do
    a. find a client process which is not busy
    b. send the document ID to this process
5. When the client process receives the document ID, it generates the chunk list for the given document, splits it by the chunk ID range, and sends the (*document ID, number of chunks IDs in the particular client range, the list of those chunk IDs*) structures to the other appropriate processes in the computational network
6. When the client process receives the data for a given document, computed in step 5, it does the following:
    a. stores the list of chunk IDs for a given document (limited to its own range of chunk IDs)
    b. stores the (*chunk ID, document ID*) pairs into appropriate buckets, into which it splits its own chunk ID space even further.
    c. adds a *document ID* to the list of documents, which are being recomputed.
7. After all documents from the step 4 have been handled, the client processes do the following:
    a. sorts the buckets generated in step 6b, and concatenates them into the (part of an) inverted index

   b. takes the old inverted index (received in step 3), deletes from it all document IDs from the list generated in the step 6c, and merges it with the data generated in the step 6a.
8. After a new inverted index is generated, all client processes send their part of the inverted index to the central process, which then forms a new inverted index for use in the step 3 of the next run of the algorithm.
9. The client processes take the lists generated in the step 6a, and for each newly recomputed document, find documents which share some chunks in their part of the chunk ID space with this recomputed document. To the process from which the data about this document has been previously received, send the information about which other documents share with this document how many chunks.
10. The client processes receive the data from the step 9 from each other client process, merge the results, and now for each document have a list of other documents with common chunk IDs, together with the number of chunk IDs which they have in common. From this information, the similarity data can be computed and/or updated the same way as in point 5 of the sequential variant of the incremental algorithm.

3.2. Notes to the algorithm

- In step 4b, we do not actually assign documents to the computing processes one-by-one, but instead we cluster the documents by hundred a time, in order to keep the computing processes busy.
- We do not need a special way of handling new, modified, and deleted documents. The information "*this document is being recomputed*" stored in step 6c, is sufficient for handling all these three cases. For example, the information about the deleted document is equivalent to the information "*this document ID has zero chunk IDs in your part of the chunk ID space*" sent to the client process.

3.3. Distributed aspects

In the above algorithm, many parts of it can be run in parallel, especially:

- The step 3 can be run on background and has to be finished only before the step 7b starts.
- The number of buckets in step 6b can be chosen so that the data of the whole bucket can fit into the 2nd-level CPU cache. Then the whole bucket can be sorted e.g. by quicksort in memory. For our purposes, 512 buckets on each node has been sufficient.
- The step 8 can be run on background and has to be finished only before the whole run ends. In fact, receiving the whole new inverted index after completing the step 10 can be interpreted as a "commit" of the incremental run.
- Should a computing node fail, the entire run can be discarded altogether, and the new run will pick up the same set of added/deleted/modified documents, plus some more.
- Provided that the configuration of the computational network is the same between the two consecutive runs, the data computed in step 8 can be speculatively cached on the client nodes, and the step 3 of the new run can then be completely omitted. When the new configuration is different (i.e. after the node failure or restore between the incremental runs), the chunk ID space is split differently in a new run, so the step 3 is still necessary in that case.

3.4. Practical results

We have implemented the previously described algorithm in the IS MU and the theses.cz systems. We use a cluster of 45 dual-core systems (some Athlon 64 X2, some Core2 Duo) as a computational nodes, and a master server with the Oracle 10g database for the table of documents and for the database of document similarities. We run only a single computational

process per host, because the servers have also other tasks such as serving the IS MU or theses.cz web application pages, and we do not want to increase their interactive latency times for those applications.

To avoid extremely large table of similarities, we keep the highest 100 similarities for each document only. With this limitation, we have about 30 millions of (*document 1 ID, document 2 ID, similarity*) rows in the table of similarities.

The computation of all the document similarities in a given document base takes about three hours, of which more than two are occupied by inserting the similarities to the Oracle database as described in the step 10.

This can be improved by creating the file with raw data, and then importing them directly to the database on the database server. However, we do not do this for practical purposes (one reason is that no process runs on the database server itself, and raw imports in Oracle cannot be done remotely, the other one is that the full recomputation is usually done only for testing purposes).

The incremental run is usually finished in 15 to 20 minutes. This time is largely dominated by steps 3 and 8, which we do no yet run on background, and we do not do the caching suggested in the section 3.3. The next longest part of the algorithm is again inserting the results to the Oracle database and deleting the old rows for the documents which are being deleted/modified/added.

For practical purposes the time taken by the incremental run is sufficient, because there are other significant delays inside the document handling system (such as converting DOC to PDF and plain text, running the bitmap-based documents through the OCR software, etc.).

4. FUTURE WORK

The real-world implementation, which has been running for about a year inside the IS MU and theses.cz systems, has still some shortcomings, which we would like to address in future:

- We have to implement the client-side caching of the inverted index between the incremental runs, as suggested in section 3.3.
- We should run the steps 3 and 8 in background.
- We are considering using some other storage engine than the Oracle database for the table of similarities. The custom solution would most probably be much faster, but it would have other drawbacks, like bigger implementation overhead for things like database joins with other data stored inside the system.

5. CONCLUSIONS

We have described a new generation of the system for finding similar documents in a large document base. This implementation has been in practical use for about a year, with previous generations being available since the August 2006. The system is almost fully distributed, can tolerate node failures, and can run on a commodity hardware.

The theses.cz system contains about 35,000 theses with full texts available, and together with seminar works, essays and other documents in IS MU compares the base of 1,200,000 documents for similarities. So far we are not aware of any other nation-wide theses archive with the built-in system for discovering similar documents. The biggest archive - Theses Canada [7] - claims to have about 50,000 theses available in electronic form, but has no system for discovering similarities.

We have proposed the algorithm which can be easily distributed, we have described its shortcomings and advantages. The most non-obvious part of the algorithm is using a hash-based

chunk IDs, which can greatly reduce the size of data we need to handle. We have discovered that even though this approach is not exact, hash function collisions do not introduce significant inaccuracy to the system.

ACKNOWLEDGEMENTS

The authors of this article want to thank all the members of the IS MU and theses.cz development team system for their support. Special thanks to Lucie Pekárková for careful proofreading and importing the text to MS Word.

REFERENCES

[1] Czech National Archive of Graduate Theses, http://theses.cz/

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

[3] Masaryk University Information System, http://is.muni.cz/

[4] Kasprzak, J., Brandejs, M., Křipač, M. and Šmerk, P. (2008). Distributed System for Discovering Similar Documents – From a Relational Database to a Custom-based Parallel Solution, *ICEIS 2008 Proceedings*

[5] Silverstein, A. (2002), Judy IV Shop Manual, http://judy.sourceforge.net/doc/shop_interm.pdf

[6] Dean, J. and Ghemawat, S. (2004), MapReduce: Simplified Data Processing on Large Clusters, *USENIX Operating Systems and Design Implementation '04*

[7] Theses Canada, http://collectionscanada.ca/thesescanada/index-e.html