WINGS: A Parallel Indexer for Web Contents

Fabrizio Silvestri[1,2], Salvatore Orlando[3], and Raffaele Perego[1]

[1] Istituto di Scienze e Tecnologie dell'Informazione - ISTI–CNR, Pisa, Italy
[2] Dipartimento di Informatica, Università di Pisa, Italy
[3] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy

Abstract. In this paper we discuss the design of a parallel indexer for Web documents. By exploiting both data and pipeline parallelism, our prototype indexer efficiently builds a partitioned inverted compressed index, a suitable data structure commonly utilized by modern Web Search Engines. We discuss implementation issues and report the results of preliminary tests conducted on a SMP PCs.

1 Introduction

Nowadays, Web Search Engines (WSEs) [1,2,3,4] index hundreds of millions of documents retrieved from the Web. Parallel processing techniques can be exploited at various levels in order to efficiently manage this enormous amount of information. In particular it is important to make a WSE scalable with respect to the size of the data and the number of requests managed concurrently.

In a WSE we can identify three principal modules: the *Spider*, the *Indexer*, and the *Query Analyzer*. We can exploit parallelism in all the three modules. For the Spider we can use a set of parallel agents which visit the Web and gather all the documents of interest. Furthermore, parallelism can be exploited to enhance the performance of the Indexer, which is responsible for building an index data structure from the collection of gathered documents to support efficient search and retrieval over them. Finally, parallelism and distribution is crucial to improve the throughput of the Query Analyzer (see [4]), which is responsible for accepting user queries, searching the index for documents matching the query, and returning the most relevant references to these documents in an understandable form.

In this paper we will analyze in depth the design of a parallel Indexer, discussing the realization and the performance of our WINGS (Web INdexinG System) prototype. While the design of parallel Spiders and parallel Query Analyzers has been studied in depth, only a few papers discuss the parallel/distributed implementation of a Web Indexer [5,6].

Several sequential algorithms have been proposed, which try to well balance the use of core and out-of-core memory in order to deal with the large amount of input/output data involved. The *Inverted File* (*IF*) index [7] is the data structure typically adopted for indexing the Web. This is mainly due to two main reasons. First it allows the efficient resolution of queries on huge collections of Web pages, second it can be easily compressed to reduce the space occupancy in order to obtain a better exploitation of the memory hierarchy [8].

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3036, pp. 263–270, 2004.

An IF index on a collection of Web pages consists of several interlinked components. The principal ones are: the *lexicon*, i.e. the list of all the *index terms* appearing in the collection, and the corresponding set of *inverted lists*, where each list is associated with a distinct term of the lexicon. Each inverted list contains, in turn, a set of *postings*. Each posting collects information about the *occurrences* of the corresponding term in the collection's documents. For the sake of simplicity, in the following discussion we will consider that each posting only includes the identifier of the document (*DocID*) where the term appears, even if postings actually store other information used for document ranking purposes.

Another important feature of the IF indexes is that they can be easily partitioned. In fact, let us consider a typical parallel Query Analyzer module: the index can be distributed across the different nodes of the underlying architecture in order to enhance the overall system's throughput (i.e. the number of queries answered per each second). For this purpose, two different partitioning strategies can be devised. The first approach requires to horizontally partition the whole inverted index with respect to the lexicon, so that each query server stores the inverted lists associated with only a subset of the index terms. This method is also known as *term partitioning* or *global inverted files*. The other approach, known as *document partitioning* or *local inverted files*, requires that each query server becomes responsible for a disjoint subset of the whole document collection (vertical partitioning of the inverted index). Following this last approach the construction of an IF index become a two-staged process. In the first stage each index partition is built locally and independently from a partition of the whole collection. The second phase is instead very simple, and is needed only to collect global statistics computed over the whole IF index.

Since the document partitioning approach provides better performance figures for processing typical Web queries than the term partitioning one, we adopted it in our Indexer prototype. Figure 1 illustrates this choice, where the document collection is here represented as a set of html pages.

Note that in our previous work [4] we conducted experiments where we pointed out that the huge sizes of the Web make the global statistics useless. For this reason, we did not consider this phase in the design of our indexer.

The paper is organized as follow. Section 2 motivates the choices made in the design of WINGS and discusses parallelism exploitation and implementation issues. Some encouraging experimental results obtained running WINGS on Linux SMP PCs are instead presented and discussed in Section 3. Finally, Section 4 draws some conclusions and outlines future work.

2 The Design of WINGS

The design of a parallel Indexer for a WSE adopting the document partition approach (see Figure 1, can easily exploit *data parallelism*, thus independently indexing disjoint sub-collections of documents in parallel. Besides this natural form of parallelism, in this paper we want to study in depth the parallelization

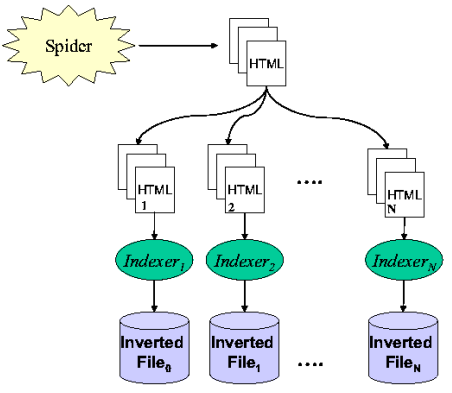

Fig. 1. Construction of a distributed index based on the document partition paradigm, according to which each local inverted index only refers to a partition of the whole document collection.

opportunities within each instance of the Indexer, say $Indexer_i$, which accomplishes its indexing task on a disjoint partition i of the whole collection.

Table 1. A toy text collection (a), where each row corresponds to a distinct document, and (b), the corresponding inverted index, where the first column represents the lexicon, while the last column contains the inverted lists associated with the index terms.

Document Id	Document Text
1	Pease porridge hot, pease porridge cold.
2	Pease porridge in the pot.
3	Nine days old.
4	Some like hot, some lite it cold.
5	Some like in the pot.
6	Nine days old.

(a)

Term	Postings list
cold	1, 1
days	3, 6
hot	1, 4
in	2, 5
it	4, 5
like	4, 5
nine	3, 6
old	3
...	...

(b)

The job performed by each $Indexer_i$ to produce a local inverted index is apparently simple. If we consider the collection of documents as modeled by a matrix (see Table 1.(a)), building the inverted index simply corresponds to transpose the matrix (see Table 1.(b)). This matrix transposition or inversion can be easily accomplished in-memory for small collections. Unfortunately, a naive in-core algorithm becomes rapidly unusable as the size of the document collection grows. Note that, with respect to a collection of some GBs of data, the size of the final lexicon is usually a few MBs so that it can be maintained

in-core, while the inverted lists cannot fit into the main memory and have to be stored on disk, even if they have been compressed.

To efficiently index large collections, a more complex process is required. The most efficient techniques proposed in literature [7], are all based on external memory sorting algorithms. As the document collection is processed, the Indexer associates a distinct *DocID* with each document, and stores into a in-core buffer all the pairs $< Term, DocID >$, where $Term$ appears at least once in document $DocID$. Buffer size occupies as much memory as possible, and when it becomes full, it is sorted by increasing $Term$ and by increasing $DocID$. The resulting *sortedruns* of pairs are then written into a temporary file on disk, and the process repeated until all the documents in the collection are processed. At the end of this first step, we have on disk a set of sorted runs stored into distinct files. We can thus perform a multi-way merge of all the sorted runs in order to materialize the final inverted index.

According to this approach, each $Indexer_i$ works as follows: it receives a stream of documents, subdivides them in blocks, and produce several disk-stored sorted runs, one for each block. We call this phase $Indexer_i^{pre}$. Once $Indexer_i^{pre}$ is completed, i.e. the stream of pages has been completely read, we can start the second phase $Indexer_i^{post}$, which performs the multi-way merge of all the sorted runs.

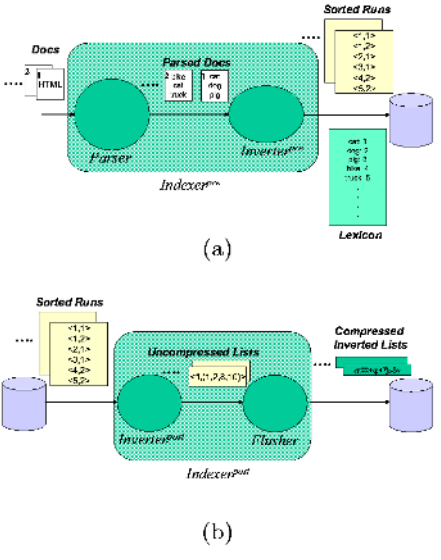

Fig. 2. Forms of parallelism exploited in the design of a generic *Indexer*, in particular of the first module (a) $Indexer_i^{pre}$, and the (b) $Indexer_i^{post}$ one.

The main activities we have identified in $Indexer^{pre}$ and $Indexer^{post}$ are illustrated in Figure 2.(a) and Figure 2.(b), respectively.

$Indexer^{pre}$ can be in turn modeled as a two stage pipeline, $Parser$ and $Inverter^{pre}$. The former recognizes the syntactical structure of each document (html, xml, pdf, etc.), and generates a stream of the terms identified. The $Parser$ job is indeed more complex than the one illustrated in figure. It has to determine the local frequencies of terms, remove stop words, perform stemming, store information about the position and context of each occurrence of a term to allows phase searching, and, finally, collect information on the linking structure of the parsed documents. Note that information about term contexts and frequencies are actually forwarded to the following stages, since they must be stored in the final inverted files in order to allow to rank the results of each WSE query. For the sake of clarity, we will omit all these details in the following discussion.

The latter module of the first pipeline, $Inverter^{pre}$, thus receives from the $Parser$ stage a stream of terms associated with distinct $DocID$s, incrementally builds a lexicon by associating a $TermID$s with each distinct term, and stores on disk large sorted runs of pairs $< TermID, DocID >$. Before storing the run, it has to order them first by $TermID$, and then by $DocID$. Note that the use of integer $TermID$ and $DocID$ not only reduces the size of each run, but also makes faster comparisons and thus runs' sorting. $Inverter^{pre}$ has a sophisticated main memory management, since the lexicon has to be kept in-core, each run has to be as large as possible before flushing to disk, and we have to avoid memory swapping.

When the first pipeline $Indexer^{pre}$ ends processing a given document partition, the second pipeline $Indexer^{post}$ can start its work. The input to this second pipeline is exactly the output of $Indexer^{pre}$, i.e., the set of sorted runs and the lexicon relative to the document partition. The first stage of the second pipeline is the $Inverter^{post}$ module, whose job is to produce a single sorted run starting from the various disk-stored sorted runs. The sorting algorithm is very simple: it is a in-core multi-way merge of the n runs, obtained by reading into the main memory the first block b of each run, where the size of the block is carefully chosen on the basis of the memory available. The top pairs of all the blocks are then inserted in a (min-)heap data structure, so that the top of the heap contains the smallest $TermID$, in turn associated with the smallest $DocID$. As soon as the lowest pair p is extracted from the heap, another pair, coming from the same sorted run containing p (i.e., form the corresponding block), is inserted into the heap. Finally, when an in-core block b is completely emptied, another block is loaded from the same disk-stored run. The process clearly ends when all the disk-stored sorted runs have been enterely processed, and all the pairs extracted from the top position of the heap.

Note that $Indexer^{post}$ as soon as extracts each ordered pair from the heap, forwards it to the $Flusher$, i.e. the latter stage of the second pipeline. This stage receives in order all the postings associated with each $TermID$, compresses them by using the usual techniques based on the representation of each inverted list as a sequence of gaps between sorted $DocID$s, and stores each compressed list on disk. A scratch of the inverted index produced is shown in Figure 3. Note that the various lists are stored in the inverted file according to the ordering given

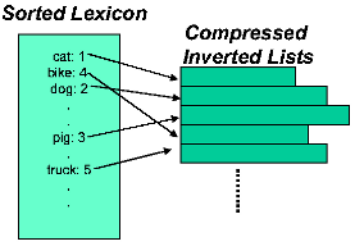

Fig. 3. Inverted file produced by the *Flusher*.

by the *TermID* identifiers, while the lexicon shown in figure has to be finally sorted according to lexicographic order given by the corresponding terms.

3 Experimental Results

All the tests were conducted on a 2-way SMP equipped with two Intel 2.0 GHz Xeon processors, one GB of main memory, and one 80 GB IDE disk.

For the communication between the various stages of the *Indexer* pipelines, we used an abstract communication layer that can be derived in order to exploit several mechanisms. In the tests we performed, since the pipeline stages are mapped on the same node, we exploited the System V message queue IPC.

In order to evaluate the opportunity offered by the pipelined parallelization scheme illustrated above, we have first evaluated the computational cost of the *Parser*, the *Inverter*, and the *Flusher* module, where each of them uses the disk for input/output.

Table 2. Sequential execution times for different sizes of the document collection.

Collection Size (GB)	Time (s) *Parser*	Time (s) *Inverter*	Time (s) *Flusher*	Tot. Throughput (GB/h)
1	3610	1262	144	0.71
2	7044	3188	285	0.68
3	11355	4286	429	0.67
4	14438	5714	571	0.69
5	18047	6345	725	0.71

As you can observe from the execution times reported in Table 2, the most expensive module is the *Parser*, while the execution time of the *Flusher* is, on average, one order of magnitude smaller than whose of the *Inverter*. Moreover, we have to consider that, in the pipeline version, the *Inverter* module will be split into two smaller ones, $Inverter^{pre}$ and $Inverter^{post}$, whose cost is smaller than the whole *Inverter*. From the above considerations, we can conclude that

the pipeline implementation will result in an *unbalanced computation*, so that if we execute a single pipelined instance $Indexer_i$ on a 2-way multiprocessors, processors, it should result in a under-utilization of the workstation. In particular, the $Inverted^{pre}$ should waste most of its time waiting for data coming from the $Parser$.

When a single pipelined $Indexer_i$ is executed on a multiprocessor, a way to increase the utilization of the platform is to try to balance the throughput of the various stages. From the previous remarks, in order to balance the load we could increase the throughput of the $Parser$, i.e. the most expensive pipeline stage, for example by using a multi-thread implementation where each thread independently parses a distinct document[1]. The other way to improve the multiprocessor utilization is to map multiple pipelined instances of the indexer, each producing a local inverted index from a distinct document partition.

Table 3. Total throughput (GB/s) when multiple instances of the pipelined $Indexer_i$ are executed on the same 2-way multiprocessor.

Coll. Size	No. of instances			
(GB)	1	2	3	4
1	1.50	2.01	2.46	2.57
2	1.33	2.04	2.44	2.58
3	1.38	1.99	2.38	2.61
4	1.34	2.04	2.45	2.64
5	1.41	2.05	2.45	2.69

In this paper we will evaluate the latter alternative, while the former one will be the subject of a future work. Note that when we execute multiple pipeline instances of $Indexer_i$, we have to carefully evaluate the impact on the shared multiprocessor resources, in particular the disk and the main memory. As regards the disk, we have evaluated that the most expensive stage, i.e. the $Parser$, is compute-bound, so that the single disk suffices to serve requests coming from multiple $Parser$ instances. As regards the main memory, we have tuned the memory management of the stages $Inverter^{pre}$ and $Inverter^{post}$, which in principle could need the largest amount of main memory to create and store the lexicon, store and sort the runs before flushing to the disk, and to perform the multi-way merge from the sorted runs.

In particular, we have observed that we can profitably map up to 4 instances of distinct pipelined $Indexers_i$ on the same 2-way processor, achieving a maximum throughput of 2.64 GB/hour. The results of these tests are illustrated in Table 3. Note that, when a single pipelined $Indexer_i$ is executed, we were however able to obtain a optimal speedup over the sequential version ($\simeq$ 1.4 GB/h

[1] The $Parser$ is the only pipeline stage that can be further parallelized by adopting a simple data parallel scheme.

vs. $\simeq$ 0.7 GB/h), even if the pipeline stages are not balanced. This is due to the less expensive in-core pipelined data transfer (based on message queues) of the pipeline version, while the sequential version must exploit the disk to save intermediate results.

4 Conclusion

We have discussed the design a WINGS, a parallel indexer for Web contents that produces local inverted indexes, the most commonly adopted organization for the index of a lrge-scale parallel/distributed WSE. WINGS exploits two different levels of parallelism. Data parallelism, due to the possibility of independently building separate inverted indexes from disjoint document partitions. This is possible because WSEs can efficiently process queries by broadcasting them to several searchers, each associated with a distinct local index, and by merging the results. In addition, we have also shown how a limited pipeline parallelism can be exploited within each instance of the indexer, and that a low-cost 2-way workstation equipped with an inexpensive IDE disk is able to achieve a throughput of about 2.7 $GB/hour$, when processing four document collections to produce distinct local inverted indexes. Further work is required to assess the performance of our indexing system on larger collections of documents, and to fully integrate it within our parallel and distributed WSE prototype [4]. Moreover, we plan to study how WINGS can be extended in order to exploit the inverted lists active compression strategy discussed in [8].

References

1. Baeza–Yates, R., Ribiero–Neto, B.: Modern Information Retrieval. Addison Wesley (1998)
2. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes – Compressing and Indexing Documents and Images. second edition edn. Morgan Kaufmann Publishing, San Francisco (1999)
3. Brin, S., Page, L.: The anatomy of a large-scale hypertextual Web search engine. Computer Networks and ISDN Systems **30** (1998) 107–117
4. Orlando, S., Perego, R., Silvestri, F.: Design of a Parallel and Distributed WEB Search Engine. In: Proceedings of Parallel Computing (ParCo) 2001 conference, Imperial College Press (2001) 197–204
5. Jeong, B., Omiecinski, E.: Inverted File Partitioning Schemes in Multiple Disk Systems. IEEE Transactions on Parallel and Distributed Systems (1995)
6. Melnik, S., Raghavan, S., Yang, B., Garcia-Molina, H.: Building a Distributed Full–Text Index for the Web. In: World Wide Web. (2001) 396–406
7. Van Rijsbergen, C.: Information Retrieval. Butterworths (1979) Available at http://www.dcs.gla.ac.uk/Keith/Preface.html.
8. Silvestri, F., Perego, R., Orlando, S.: Assigning document identifiers to enhance compressibility of web search. In: Proceedings of the Symposium on Applied Computing (SAC) - Special Track on Data Mining (DM), Nicosia, Cyprus, ACM (2004)