Massachusetts Institute of Technology
6.824 Distributed Computer Systems
T. Alatovic, U. Mobin, G. Snyder, R. Zhang

December 1, 2001

Fast Distributed Index Generator

Abstract

We present design and implementation of fast distributed index generator for a cluster of workstations in a small local area network. The index generator takes in a set of text documents and generates, for every word, a sorted list of all the occurances of that word. We focus our design efforts on three important design goals for a function of the index generator: scalability, performance, and availability.

1. Introduction

Index generation lies at the heart of most efficient search engines. Generally, an index generator takes as input a set of documents and generates an index containing the locations of every meaningful[1] word within the documents. The index generator we present uses the sort based inversion algorithm to generate the index. The computational task of index generation is distributed on a per-document basis across a cluster of machines; the primary emphasis being the speed of index generation.

The system has been subdivided into three subsystems: index creation; document distribution; and query subsystems. More information is provided on their specific functionality in the design section.

2. Related Work

Extensive research has been done in the area of efficient index generation and querying. The index generation and querying techniques used in this paper is largely based on works presented in [1]. However, the materials presented in [1] are mainly designed for single machines rather than for a distributed environment. The main focus of our research is therefore, adapting efficient single machine indexing and querying techniques to a cluster of workstations distributed over a small local area network. The index generation scheme used in our system is a distributed version of the sort-based inversion algorithm. The advantage of this algorithm is that it efficiently manages the use of the main memory to achieve high scalability.

Distributed index generation and querying boil down to efficient sorting algorithms. Therefore, the NOW-sort [2] system contains much relevant information to our research. NOW-sort is a high performance distributed sorting system using a network of

[1] From the query's perspective. For example, some index generators might ignore words like "the" and "a" because they do not have much significance from the searching-point-of-view.

workstations. We try to address many of the same issues as NOW-sort such as how to divide up the job, utilizing efficient memory management schemes, and achieving efficient disk utilization.

3. Design

We focus our design of distributed index generator on the following three goals in the order of importance:

(1) scalability
(2) performance
(3) availability

We choose to focus on these three goals because we believe they are most important for running a reliable high-performance index generator that can be used for indexing large amounts of text available on the Internet today.

We also choose not to address the issues of security and portability. The reasons are twofold: (1) we have limited design and implementation time, and (2) these issues are not of primary importance in a cluster of workstations in local area network.

3.1 Scalability

Problem of index generation on large inputs (hundreds of gigabytes) requires large amount of computational resources that are typically unavailable on a single machine. As such, index generation problem requires resources of multiple machines and calls for distributed architecture. For performance and fault-tolerance reasons we decided to implement our system on a cluster of workstations in a small local area network.

We decided to treat all nodes in the network equally. There is no central server that routes the requests and manages the load. This makes a load distribution difficult but eliminates central registry, a bottleneck and a single point of failure in the system. Figure 1 below shows the first version of our distributed index generator, which is already implemented and fully functional.

We currently have a completely functional first version of the system. The current modules of the system are: index creation module, document distribution module, and the query module.

The document distribution module hands out documents to individual servers for indexing while maintaning a document-to-server table. The local query procedure on each server searches the index on that particular server. However, all servers know about their peers and have the ability to call each other's query function. Currently, the query software (client-side) queries any server in the cluster for its list of peers and then uses the list to query each individual server. The individual results are then consolidated on client side and presented to the user.

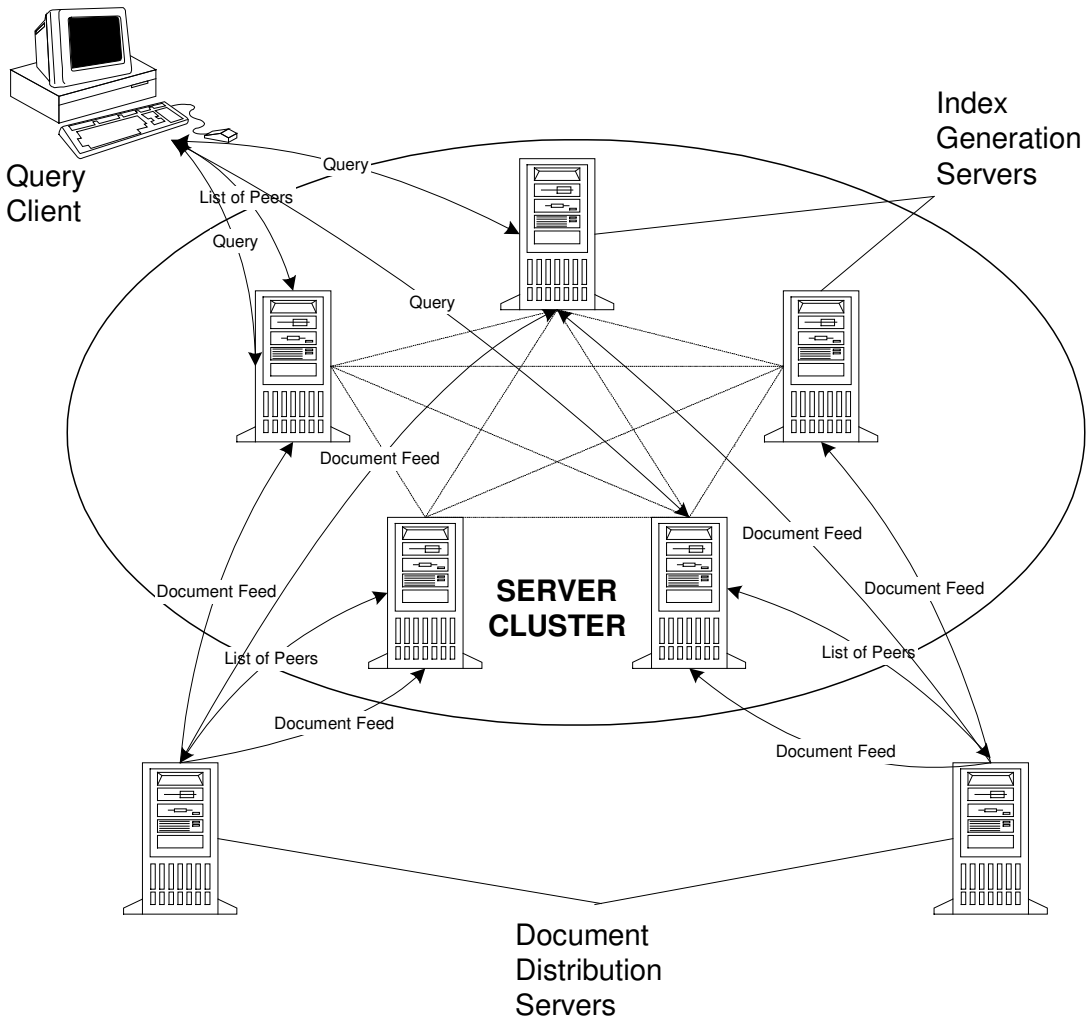

Figure 1 – Distributed Index Generator

We choose to distribute the index load by document and not by words. The advantages of per document distribution are (1) simpler algorithm for document distribution, and (2) distributing the query load for frequently occurring words across all servers in the cluster,. The disadvantages of per document distribution are (1) that query client needs to query all servers in the cluster in order to gather complete results, and (2) that it doesn't allow for fine granularity in distributing the indexing load if documents are large. However, the assumption of our design is that workstations are running in a fast inter-machine network, which mitigates disadvantage 1, and that we are primarily concerned about Internet workload with small files, which mitigates disadvantage 2.

3.2 Performance

In this section, we present parts of the system with most significant impact on the performance of the system: (1) index generation algorithm for a single machine, and (2) management of distributed computational resources.

3.2.1 Indexing Algorithm

Sort based inversion is one of the methods that are relatively easy to implement. But more importantly, it has good properties. Because sort based inversion sorts the index terms one batch at a time, it allows us to precisely control the amount of main memory the algorithm can use. Sorting all entries in a batch that fits in main memory avoids the high overhead of virtual memory disk paging.

Each node in the system can be individually customized to fully utilize the available memory to achieve maximum performance. This method is scalable because the size of the input does not affect the memory requirement. Moreover, the performance of the algorithm is acceptable. The trade off is that this method has a higher disk space requirement than some of the other methods because temporary storage is needed for the merging process. We favor this method despite its higher storage requirements for its ease of implementation and flexibility in memory usage. For implementations, which focus on lower storage space, compressing versions of the sort based inversion algorithm are available [1].

3.2.2 Management of Distributed Resources

Management of distributed computational resources is a key to building a successful distributed system. Inefficient use of distributed resources can result in under-utilization of the available system capacity and create bottlenecks on performance. Therefore, we focus on creating efficient resource management policies resulting in evenly distributed workload over index generation servers.

In our system, a document distribution server (DDS) has access to the documents to be indexed. The DDS splits the entire collection of documents into smaller fixed size blocks of documents and sends one block to each index generation server (IGS) in the cluster. It is important to note here that the DDS speaks to each IGS in a separate thread so it does not block until an IGS sends a reply before sending documents to other IGS's.

Upon receiving a block of documents, the IGS performs the sort based inversion algorithm on the documents locally, without any communication to other nodes. Since each node knows precisely how much main memory it has, the index generation algorithm on each node can be custom configured to fully utilize the main memory to avoid virtual memory disk paging.

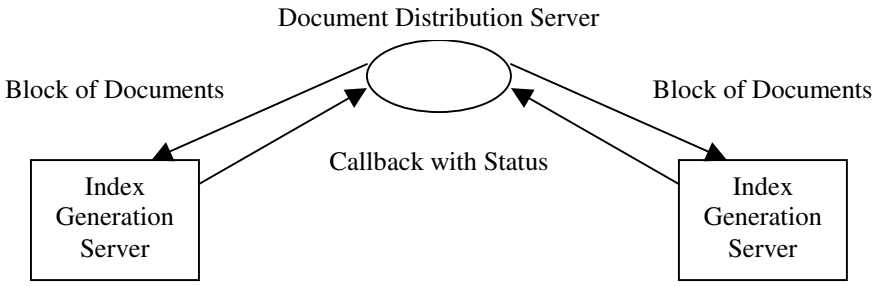

Figure 2 – Resource Management Scheme

When the IGS finishes generating index for a block of documents, it issues a callback to the thread in the DDS that sent the documents. The callback signals success or any error to the DDS, which then either send the IGS a new block of documents to process or try to recover from the error. In the event that the IGS is running out of resources such as disk space, it can signal the DDS upon callback so that no more documents are assigned to it.

Initially, we considered having IGS's reporting various resource statistics such as disk space and CPU speed to the DDS and have the DDS decide how to balance the load across the IGS's. This design turned out to be problematic. One being that a large amount of computation is placed on the DDS, so the system might run into scalability problems as the number of IGS increases. The other problem being that resource management is inherently difficult for the DDS to perform. Because there is no continuous status feedback from the IGS, resource information is simply gathered in the beginning and the DDS can only make a best estimate on how to balance the workload. Without the feedback, a fast machine might be under-utilized while a slow machine is over-utilized. Worse yet, a node might run out of disk space while indexing is in progress without the DDS knowing.

The current resource management scheme addresses all these issues by creating a continuous feedback loop between the DDS and the IGS. In doing so, the resource management tasks are mostly shifted to the IGS. Also, because each node can make the most informed decision on how much workload it can handle, resources in this distributed environment are used more efficiently. In this system, load balancing is achieved by pushing documents to the nodes as fast as they can successfully process them.

4. Robustness and Availability

In this section, we present the strategy for maintaining robustness of the system and high availability in the events of failing nodes and communication links. Later, we will consider the implications of this strategy for the scalability of the system.

In the first implementation of the system, our per-document indexes are distributed among the index-generation-cluster machines. Although this approach results in a very fast speed of query generation, it has the disadvantage that every machine in the index-generation-cluster needs to be queried to obtain the complete results. This has negative repercussions on the scalability of the system. Without the notion of replication, this results in a notion of availability where a query returns only partial (but correct) results in the event of node failures.

We would like to take our existing notion of availability a little further by considering a strategy, which results in correct and complete results being returned even in the event of the failure of multiple nodes. Of course, some form of replication is a must for such a notion of availability.

4.1 Index generation cluster

The index generation cluster receives documents from the document distribution server and each machine in the cluster generates an index for each document it receives. This was the extent of the first implementation's functionality. In the most basic form, query works as follows: the query client queries any server in the cluster for its peers and uses the list thus obtained to query every server in the cluster. As far as availability is concerned, this means that the failure of a node results in the loss of the index information pertaining to a subset of the documents. Also, if the server being queried is down then no query results are obtained. This latter problem can be easily solved by querying a generic name like query.cluster-name.com and allowing the underlying naming infrastructure (Domain Name System) to resolve the name into any one of the list of the addresses of the index-generation-cluster machines. However, there are problems with this approach. First, we would rather aim for complete and correct results even in the event of the failure of a reasonable number of nodes. Second, the fact that every server in the index-generation-cluster needs to be queried before the final results may be compiled does not yield a scalable querying solution.

4.2 Replicated Filesystem Cluster

The distributed index generator shares many properties with a distributed filesystem in terms of its availability requirements. Therefore, we are considering an approach to make the system more scalable and available by storing the index information on a distributed filesystem. This would entail a cluster of machines hosting a distributed replicated filesystem. The replication policy will depict the severity of node failures that we are willing to handle without a loss in results.

This works as follows. The document distribution server distributes documents to the index-generation-cluster machines as before. Each machine generates, on its local store, a per-document index of word occurrences, as before.

However, the distributed filesystem cluster stores a per-word index such that for each word, there is an index of documents that contain that word. The index-generation-cluster machines use the per-document indexes they create to augment the indexes stored on the distributed filesystem. Once the indexes on the distributed filesystem have been updated, the ones on the index-generation-cluster machines can be discarded. Thus, these machines have low disk storage requirements as the actual index information is now being stored on the distributed filesystem cluster.

In this new scheme, querying does not operate on the index generation cluster but instead on the distributed filesystem. The filesystem is queried for the word requested and the index for that word is used to return the results. This results in a much more efficient and scalable query mechanism.

In the case of a failure of a node in the index generation cluster, the load is automatically rebalanced by the document distribution server. However, since no permanent index information is ever stored in this cluster, the recreation of parts of the index become

irrelevant resulting in an inherently more robust system. The replicated filesystem aims to ensure that all index information is available at all times with very high probability.

The index may be organized as a file per word and these files may be stored in a directory hierarchy on the filesystem in much the same way as user home directories are organized and stored on very large systems. These directories may be mount points across various filesystems on multiple machines in the filesystem cluster.

Since our system has similar availability requirements as a distributed and replicated filesystem, layering it on top of such, in the form mentioned above, will allow us greater reliability and robustness for querying, and will make the system as highly available as provided by the underlying filesystem.

5. Measurements

We tested the first version of our system on a number of machines with different numbers and sizes of documents. We used sort based inversion algorithm for index generation and experimented with the maximum buffer size - the maximum amount of data that can be stored in memory at one time.

The following two tests were performed on a number of machines. The first test consisted of 100 random small text documents, whose sizes ranged from 24 bytes to 882 bytes. The total size of these documents was 47,682 bytes. The second collection of documents consisted of 100 random large text documents, whose sizes ranged from 489 to 90,814 bytes, for a total of 4,241,006 bytes.

These documents were generated randomly by a random document generator. The generator read in an 869,227 word dictionary and then created documents with a random number of words chosen at random from the dictionary. The number of documents and maximum number of words could be specified. The small collection of documents was generated with a 100 word maximum and the large collection was generated with a 1000 word maximum.

Our heuristic was how fast an index can be generated on these documents. We calculated the time that it took from starting to generate the index until the time when the last computer finished creating the index. We tested our index generator with a max buffer size of one-half megabyte (524,288 bytes).

Once the index was generated, a sample query was run on a random word from the dictionary. This word occurred once in the collection of small documents and multiple times in the large document set. This test was done to ensure that the index generator performs correctly and the right data is sent to all machines.

# of machines	Small Documents Index Time (s)	Large Documents Index Time (s)
1	11	N/A
2	9.5	425
3	8	289.5
4	6.5	220.5
5	6.5	170.5

Table 1 – Results of Index Generation

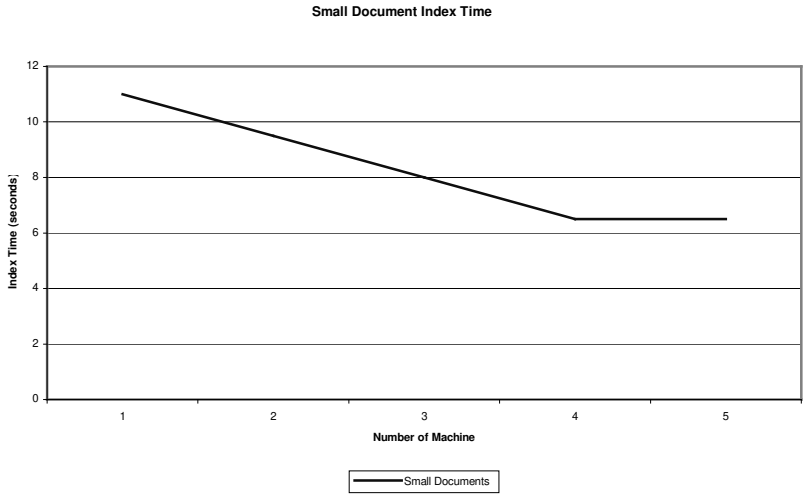

Figure 1 – Small Document Index Time

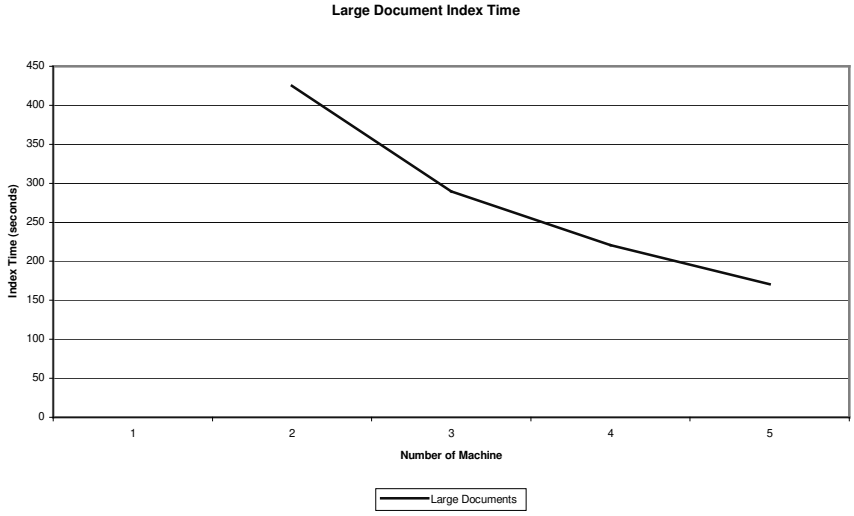

Figure 1 – Large Document Index Time

As figures 1 and 2 show, as the number of machines increased, it took less time to create the index, as expected. Looking at the small document index time, it appears that our index generator hits a peak level. This happens when the network bandwidth is the limiting element. When indexing the set of large documents, the index generator improves by over 20% each time a new computer was added.

In the situation with one machine and the large document set, the Java RMI, which our source was based on, overflowed and could not complete the test.

We used a one-half megabyte max buffer size to simulate that the data was larger than it really is. If we increased the max buffer size by two, we could double the size of the documents without affecting index generation time, assuming that the network is never the bottleneck. Today's servers can have a max buffer size from anywhere from 256 megabytes to 1 gigabyte. Our small documents are therefore relative to 23.3 megabytes, on the low

end, and 93.1 megabytes on the high end. The large document set is relative to 2,070 megabytes on the low end and 8,283.2 megabytes (8.09 gigabytes) on the high end.

6. Conclusions

A fast and distributed index generator is important for applications involving indexing of very large amounts of textual data and storing and generating the indexes. The distributed nature of the index generation allows the ability to index more content in less time than is possible through the computational resources of any one computer.

As our results show, the distributed index generator can generate indexes on large documents with increasing performance as the number of machines in the cluster is increased. We have used sort-based inversion to generate indexes rapidly on a per-document basis. In addition, we have considered the query-side scalability repercussions of this setup and considered augmentations to the system which involve appending per-word indexes from the document indexes we generate. We have realized the similarities between the reliability and availability requirements of our per-word indexes and have considered an approach where the index information is layered on top of a replicated and distributed filesystem.

References

[1] I. H. Witten, A Moffat, and T. C. Bell. Managing Gigabytes, Compressing and Indexing Documents and Images. *Academic Press*, 1999, pages 153-261

[2] A. C. Arpaci-Dusseau, R. H. Arpaci-Desseau, D. E. Culler, J. M. Hellerstein, D. A. Patterson. High-Performance Sorting on Network of Workstations. *SIGMMOD*, May 1997.