FDIG - Fast distributed index generator

December 6, 2001

Abstract

This paper describes the architecture of a distributed index generator and a query system, which are the major components of a large-scale Web search engine that can index more than a billion pages with terabytes of data. We present an index generator that distributes the task over multiple nodes to reduce the wall-clock cold startup time proportionally to the number of nodes – otherwise indexing on a single host can take time in the order of weeks. The major problems discussed are schemes for partitioning the input and output data on the nodes using consistent hashing, load balanced processing, finding efficient local node indexing algorithms, and efficient query processing mechanisms on the distributed index.

1 Introduction

Indexing and query are the basic operations of a Web search engine. With input dataset sizes in terabytes the performance of both operations becomes critical – the cold startup indexing of a single host system can take computation time in order of weeks. Distributing the computation over multiple nodes reduces wall-clock time for the system startup.

The index generator is the core of our system, but it is important to provide efficient interfaces to its input and output — the storage and query systems. The input to the index generator is a pre-processed set of the text contents of the original files (HTML or other). We assume the input data cannot be stored on any one node prior to indexing, and thus we provide a distributed file system storage. This storage can be used as a cache or even as a primary storage for the documents, like CFS [4].

The output of the system is directly used only by a query engine that is build on top of our index generator system. An interface that allows conjunctive queries, and others restricting queries like word distance proximities is necessary. This requires collection of word positions for each word occurrence in a document. Since the input can be reconstructed from this data, the produced output is comparable in size to the input; thus we need distributed storage and a partitioning mechanism for the output.

The described index generator system can be implemented as a distributed, heterogeneous and highly scalable peer-to-peer system. In that case it would be very important to make it fault-tolerant and completely decentralized. However, we'll concentrate in this paper on the means used to improve cold startup and query performance for deployment in a cluster of workstations.

2 Related work

There is extensive literature on single node index generation used in text retrieval systems [7],[13], [3].

An *inverted file index*, as explained in [13], consists of two parts: a set of inverted lists for each queryable term consisting of pairs of file identifiers and offsets; and a search structure for mapping from a word to the location of its inverted list.

Indexing design comparisons between inverted files and *signature files* are found in [8]. They paid particular attention to query evaluation speed and space requirements and show that inverted files are much cheaper to build than signature files, since signature files are much larger than inverted files and they are also more expensive to update. Zobel [13] proves that still uncompressed inverted files can consume a space, potentially several times as the data they index, and propose efficient compression methods of index files. These compression costs are repaid by reductions in disk I/O [5].

A bitmap [7] is a very simple indexing structure: for every term in the *lexicon* — the set of all words in the collection — a bitvector is stored, each bit corresponding to a document. Bitmaps are fast and easy to use, but extravagant in storage. For a text of N documents and n distinct words, a bitmap occupies Nn bits. This number can easily grow very large. The conclusion in [7] is that compressed inverted files are the most useful method of indexing a large collection of variable-length text documents.

The search engine prototype proposed in [2] describes the need of clever index generation to efficiently handle large amounts of queries per second. The bottlenecks in the single computer system used in Google [2] were memory access, disk seek, throughput, capacity and network I/O, while CPU utilization was often low.

Bailey and Hawking [1] proposed a design based on inverted file indexes and dictionaries kept on secondary storage. Their inverted file indexes are kept in primary memory along with the raw text. The form of index adopted is simply a list of byte pointers to the start of each indexed word. This list is sorted so that the words referenced are in lexicographic order. They use "super dictionaries", created by merging the dictionaries for each sub-collection. The part of a super dictionary associated with a processing node compromises two parts. The first is a complete list of all unique words in the combined sub-collection chunks for the node, and the second is a record that includes the frequency of occurrences of the word in each sub-collection chunk, and the relevant offset into the sub-collection chunk's index files.

Nevertheless, distributed systems are expected to be the most efficient solution for indexing [2]. Hawking [5] approaches the problem of indexing using a cluster of workstations, and used compressed inverted files of (document-number, offset) pairs, similarly to [13]. Each entry in the dictionary of all words occurring in the indexed collection points to the start point in the compressed index of the block of the position pair of that block. Their index algorithm requires a number of passes though the input. First pass: builds table of bundles, a table of documents and a hash table of all distinct words. Subsequent passes through the input file write successive windows of the compressed index, which is mapped into the process virtual memory using the Unix `mmap(2)` call. The number of passes required depends upon the ratio of compressed index size to the available physical memory. Each pass adds cost of an additional sequential scan of the tokenized input.

3 Design

The described fast distributed index generator (FDIG) generates an inverted file index that records document number and word offset pairs for each word occurence. The major design decisions consider a flexible local node indexer, schemes for data distribution and their effect on distributed query performance and load balance.

3.1 Local indexer

A major factor in the overall indexing performance of the distributed system will obviously be the performance of the local indexer. In our initial measurements we found this operation to be disk bound. Therefore, one design goal of the system was to minimize disk operations especially disk seeks.

The input to the local indexer is a list of sources (files/URLs). Each source must have an ID (32-bit) that is unique in the local system and is provided by the storage subsystem. It has to provide an index organized so that for each word the vector of (file ID, offset) pairs can be accessed with a small number of disk seeks.

The following sections discuss approaches to efficient in-memory indexing, methods of combining in-memory indices and their expected performance.

3.1.1 Dictionary

All distinct words in a data collection form its *dictionary* or *lexicon*. Keeping the dictionary in memory is important for both fast indexing and queries so that disk seeks are avoided [12]. Experiments in [13] show that searching using dictionaries increases speed and space efficiency, and runs orders of magnitude faster than brute search and requires less memory than other pattern-matching data structures. Google [2] in 1998 used a 14 million word dictionary in a hash table taking 293 MB of memory, which they claim is quite cheap for their dedicated system. However, even today not many systems have that much memory. In our efforts for higher scalability, we were looking for an efficient representation, that can also be offloaded to disk when ranges of words are processed. (Other subsets - like muli-level storage will involve frequent disk operations). Ranges of words can easily be specified in tree structure representations, so they can be divided in separately processable pieces, and are easy to merge. We explore in the next section lexical trees. Balanced trees are suggested [8] where B+ trees are used for storing the dictionary.

Our concern with a tree algorithm was that it will have much worse cache locality when compared to a hash function, since we have to traverse (random) paths in the tree. So trees may be slower, but since CPU time is usually not the bottleneck, the overall system performance should not change. That was confirmed in our prototype which used lexical search tries which provide the desired properties.

3.1.2 Merging indices

Once the memory used by the indexer is about to exceed a predefined limit (some factor of available physical memory), the partial results from memory can be stored as a sub-index on disk. Indexing then continues with a new sub-index in memory. This operation is repeated until all input is processed. The result of indexing then will be a set of sub-indexes. Since we want to provide efficient queries, we want to produce one large index at the end so that only one resultset has to be looked up. The next step then will be merging of sub-indexes produced by the first stage. The structure of the index files allows merging that requires just one pass through each index file.

In addition, our index generator will also allow incremental updates, i.e. after the initial dataset is processed, updates to the dataset will be reindexed without a full index generation. Merging is not necessarily done on the server that produces the sub-indices, thus in schemes in which large sub-ranges of the whole index are collected and merged at each node.

Merging n indices and their dictionaries can be done in linear time with small memory requirements. A naïve merge requires $2NDX$ disk space, but if disk space is at premium it is possible to merge "in-place" using only NDX disk space and still in linear time [6, 10].

3.1.3 Indexing design comparison

Here we compare the methods proposed in [5] and the described method of merging smaller indices.

INP - size of the input data collection

NDX - size of the corresponding inverted index

MEM - available physical memory

$n = \frac{NDX}{MEM}$

I/O operations for a multiple pass method similar to the one in [5]

$I/O = INP + nINP + n\frac{NDX}{n} = (n+1)INP + NDX$ — read the input to construct the dictionary; then for each of the n passes read the whole input and write the window in the index file

Merging indices as we showed in 4.4.1 is

$I/O = n(INP/n + NDX/n) + (nNDX/n + NDX) = INP + 3NDX$ — read a part of the input and write the corresponding index; read from all indices and write the merged index.

The size of the compressed index representation is slightly larger than the compressed input size, but it can be optimized to be about the same (theoretically they should be the same because they have the same information content). Assuming $NDX \simeq INP$ then $n \simeq \frac{INP}{MEM}$ and since the data collection size is known, n is a function of the available memory only. Hence, the more appropriate method can be adaptively chosen before for each node, i.e. multiple passes only if $n \leq 2$. Practically this can be observed, because limitations of current operating systems and file systems restrict the maximum filesize, often to 2 or 4 GB (ext2), while systems with 1 GB of memory are not a rarity. We used such a system, but we argue that this should be considered peculiarity of the current state and should not be the general case. We have implemented the index merging approach in our current system.

3.2 Distributed query processing

The efficiency of distributed query processing is highly related to the distribution of the index across the nodes: either according to the input or according the inverted index. The first methods simply uses the trivial partitioning: the index for the data collection stored at a given node is also stored at the same node, however, all nodes have to be queried to form the result. Strictly speaking, it does not have to be partitioning if we allow overlap between node data collections so to provide redundancy and fault tolerance.

The inverse method is to stripe the inverted index generated at each indexing node across the query nodes - another consistent hashing ring where hash is over a fixed length prefix of the words. A query for a single term in this case is then directed to the node that matches the prefix hash.

For deployment in a small cluster the consistent hashing implementation can simply be according to a table of all nodes, but for a scalable decentralized distributed system the system can use Chord [11].

The second method is apparently more efficient for one term queries, and similarly for disjunctive multiple term queries which are simply unions of the one term queries. However, querying all nodes does not have to take longer than querying one node. Indeed, a query directed to all nodes does not require each node to have a complete table of all nodes, or to flood the network. Instead it can be implemented as an extension of the Chord lookup primitives to issue recursive queries to all nodes, so that the query will take $O(logn)$ hops to arrive at each node, which is the same as the number of hops to query just one node. Thus the network latency when processing a single term query is equal for both methods.

High precision queries, however, are usually conjunctive or proximity restricted queries. Experience from TREC queries [5] shows successful queries have orders of 50–500 query terms to find the relevant documents, while very short queries return large result sets of irrelevant ones. We limit the current system to content based searches only, i.e. without using Web link data as [2], and so we try to efficiently handle queries of ten or more terms.

When conjunctive multiple term queries are issued, querying all nodes now appears advantageous: very few matches will be reported to the issuer because the intersection of the result sets of documents containing each of the query terms is much smaller than their union. The striped index on the other hand will return a large result set for each of the terms, only to have their intersection computed at the issuer. The size of the intersection of result sets is much less than the size of their union, hence much less data has to be carried for multiple term queries with the first scheme and less bandwidth will be consumed. Furthermore, when partial failures occur there will be relevant partial results, even in the case of network partitioning [4].

A strong point in favor of the striped partitioning scheme in LAN environments with low network latency is that the queries for the different terms will be routed to different nodes making a few (usually one) disk seeks to read the resultset. In contrast, a single host looking up its local index will have to make a disk seek for each of the terms in the query, which will drive down the system response time.

Overall, keeping the index for each collection on the node on which it was generated, is probably preferable in a WAN environment. In addition, local indexing is a necessary step before striping, since index subranges can be efficiently transferred between the servers only in large batches. A hybrid system that keeps both indexes may use an empirically found threshold to decide which method to use for a particular query - e.g. a striped index for queries of less than 10 terms. The cost of this flexibility is doubling the size of the index storage. We have evaluated only query times for locally kept indices.

3.3 Load balance

The overall system time for index generation is the time taken before the last node finishes. Assuming all nodes start at the same time, the difference in processing time for the nodes can be attributed to workstation speed and load; dataset size; and indexing difficulty of each collection. Equalizing the data size only has proven not to be very effective: Hawking [5] reports that the time to index on equivalent systems datasets that are equal in size can vary by 50%. Using distributed hashing for document storage distribution [4] and later indexing of the stored collection improves load balance in all forementioned terms.

We do not take explicit provisions to compensate for the variable speed of processors, although we may employ the notion of *virtual node* from [9], as used in [4] to run multiple FDIG instances on one real server.

Since we use a hash to map documents to nodes, each node will have an uniformly distributed collection of documents. The randomization will improve the load-balance in terms of size, and the system will also be more robust to indexing difficulty skew. The adverse effect of this distribution is that the dictionary size of the randomized collections node is likely to be larger than the dictionary size of collections of documents from the same domain (domain of knowledge or of Internet hosts) since the last will likely share a smaller common vocabulary. We believe that controlling the balance skew is more important for improving the overall system initialization time.

Query engine can be running on a host that is not an indexer node, but a more scalable approach would be to run a query engine on each of the nodes. Queries will then be randomly distributed to the servers, (when using Chord a hash of a random key will be directed to a random node). In a production system, other standard optimization mechanisms like query caching may be added to the system.

The system does not need any synchronization points, and nodes can be added and removed at any time supported by the Chord primitives and may start indexing input data or reply to queries.

3.4 Storage distribution

A consistent hash of the filename (URL) is used to produce a corresponding node to store the data in its data collection. The node storing a data collection will also index it. There is no centralized global naming for the files: the virtual file ID's is composed of a pair of FDIG node identifier and a local file ID.

The current prototype is working in a LAN of controlled workstations, so consistent hashing can be implemented simply with a shared static node table. However, the system can be deployed as a WAN peer-to-peer index generator if its storage system is based on the CFS system from [4]. The store module then will be simply a CFS client. However, some modifications of the CFS backend or the indexer will have to be made because the block oriented design of CFS will be inefficient for a non-striped local index: a conjunctive query cannot produce a matching document set for the locally stored data. Thus an additional intersection operation is needed; a striped index query requires only intersection and will not be affected.

If the input to the system is exposed as a filesystem interface then that will make it transparent to users to put data to be indexed. That way we can run a gatherer with a working directory in the mapped filesystem and its output will be immediately distributed to an indexing host. We may also implement in the future a distributed gatherer module to traverse and collect the files for the dataset, given initial lists of URL starting points.

3.5 Compression

Uncompressed inverted files can consume considerable space, and naïve representations might occupy 150% of the space of the text itself. For example, in typical English prose the average word contains about five characters, and each word is normally followed by one or two bytes of white-space or punctuation character. If word occurences are stored as 32-bit document numbers, and supposing that there is no duplication of words within documents, there might thus be four bytes of inverted file entry for every six bytes of text. When a four-byte "word offset within document" field is added to each pointer, the index consumes eight bytes for roughly each six bytes of text. More generally, for a test of N documents and an index containing f pointers, the total space required is f*[log N] bits, provided that pointers are stored in a minimal number of bits.

Compression should be used both for the text and for the index to save disk space and to reduce overall retrieval times at a comparatively small increase in CPU utilization (even with compression the indexing system remains I/O bound). The input dataset can be compressed with any text stream encoding algorithm; we were plan-

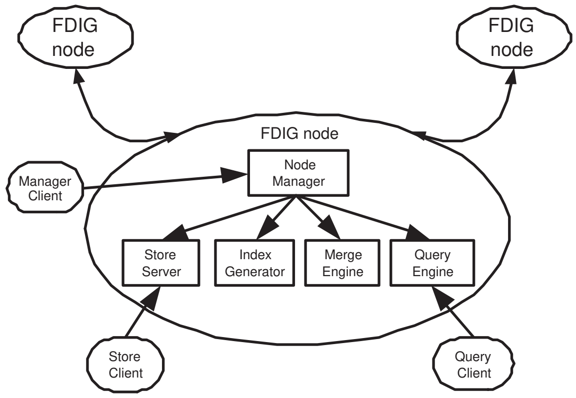

Figure 1: FDIG Architecture

ning to use the LZ77 based library `zlib(3)`, we could also use Huffman word coding or any of the other methods mentioned in [14]. The inverted index, however, has to support random access, so it should be compressed using the Elias gamma coding both for document IDs and the offsets within each document, similarly to the strategy described in [5].

4 Implementation

4.1 Node manager

The node manager is the only application started directly by the user on each host. It subsequently starts the other local services, identifies the other nodes, and keeps the consistent hashing tables. In the current prototype the information about the other nodes is in a shared static table, however an interface to Chord was supported.

The manager client talks to the local node manager and provides status information. Furthermore, it gives manual control over the system. The node manager allows a single node to broadcast phase change and requests for restart of services (including itself) to the whole system, this is enabled only when trusted distributed management is applicable. It allows a single administrator to control the whole system. It was also invaluable for incremental development, so that up to date services are started on all nodes.

4.2 Store server

The store server is the process that collects the data from the store clients. The store client usually talks to a store server on the local host, but could run on any other node. The contacted store server is given a pair of (filename,contents) which it hashes to an FDIG node and reroutes the request to it. In the prototype this task was not fully automated: the store server had minimal functionality and provided just hashing, so that the store client could send all files hashed to a node in a single batch.

4.3 Index generator

In the prototype we represented the dictionary in a search trie. One of the goals of our research was to show that development effort can be significantly lowered when simpler techniques are used. That is why we did not put much efforts in applying all known techniques for getting a single node server running faster. Specifically, we did not use the compression techniques described in Section 3.5

4.3.1 Definitions

Let F = { F_1, F_2,..., F_n } be a set of input files.
Let w_{ij} be word number j in F_i.
Let A = { $\alpha_1, \alpha_2, \ldots, \alpha_m$ } be the alphabet.
Let M_w be the number of words in F.

4.3.2 Search trie structure

Let R_t be the root node of T. Let N_t(k) be son number k of node N_t. ($1 \leq k \leq |A|$) Each node N_t in T corresponds to word W in the following way:

If W = $[\alpha_k]$, then N_t R_t(k). If W = $[\alpha_{k1}\alpha_{k2}\ldots\alpha_{kjWj}]$ and N'_t corresponds to word $[\alpha_{k1}\alpha_{k2}\ldots\alpha_{kjW-1j}]$, then N_t $N'_t(k_{jWj})$.

Let |T| be the number of nodes in T.

The index consists of two files:

FileT (serialized form of the lexical trie) consists of |T| records.

Each record corresponds to node N_t and word W. A record has the following fields:

- Number of times that W occurred in F
- Offset in FileP where all occurrences of W is stored
- Bitmap B = ($b_1 b_2 \ldots b_m$). b_k true if and only if there is a son number k of N_t

Record that corresponds to word W is followed by records that correspond to words having W as a prefix. Therefore, each sub-tree of T is placed contiguously in FileA (*).

FileP consists of M_w pairs: (file ID, word ID).

If a lexical trie T does not fit in the physical memory, additional, super-index file FileT' will be used. This file will hold sub-tree T' representing only the upper levels of T. Each node N_t in T' will hold offset in FileT that corresponds to record of N_t.

Some immediate observations:

- If only FileT and FileP are used (T in memory), only one read in FileP will be required in order to perform one-word query.
- If FileT, FileP and FileT' are used (T' in memory), one read from FileP and one read from FileT will be required in order to perform one-word query.
- Because of (*), merge of indexes will require just one pass through each index file without the need to hold any of the files in physical memory.
- FileT' can be easily constructed from FileT.

4.3.3 Indexing stage

The elementary operations of the indexer are: reading the sources, building the lexical tree T and for each word W, remembering the source position identified by (file ID, word ID) pair.

Once the memory used by the indexer is about to exceed a predefined limit (a factor of available physical memory), the partial results will be stored in (FileT_k and FileP_k). Afterwards, we continue to process the input starting with a new lexical tree. This operation is repeated until all input is processed. The result of this stage will be a set of sub-indexes: (FileT_k, FileP_k) pairs.

4.4 Merge engine

The merge engine is a separate component started by the node manager typically immediately after the indexer is done. It may also be run synchronously with mergers on all other machines, or if indexing is incremental at some time intervals. It could also be interleaved with the indexer if there is good hardware support for that – dual processor and disk arrays. However, in our experiments it was started only after indexing is complete.

4.4.1 Merging stage

The current implementation uses a straightforward merge algorithm to merge the sub-indexes produced by the first stage. We do it by merging { FileT_k } to one FileT and merging { FileP_k } to one FileP. The structure of the index files allows merging that requires just one pass through each index file. Merging was limited by the forementioned ext2 restrictions, although it wouldn't be very hard to go around it.

4.5 Query engine

The query engine provides a simple query interface for single word queries as well as multiple word conjunctive and phrase queries (the last are the justification for keeping word offsets in the inverted indices).

4.5.1 Single word query

Query in a single inverted index proceeds as follows: given a single query word, we traverse the lexical trie taking the appropriate branch at word position. Once we reach a leaf node we check if are at the end of the looked up word, if so we have a match and we may retrieve the entries for that word. Otherwise, we don't have a record for that query word in our collection. If the index `posArray(fileid,filepos)` is in memory we simply return the pairs. When the index is on the hard disk, in FileP, we use the the word's entry start offset from FileT and then look up the corresponding records from FileP which contain file IDs and word positions. With this scheme, the query results can usually be retrieved with only one disk seek.

4.5.2 Multiple word query

A multiple words query starts with single word queries to find the size and locations of the records for all terms and then reads the data from the index file on disk. If this is done asynchronously, a smart disk driver could optimize the order of the read requests. After all position vectors are in memory, their intersection is computed: for each match of the word with the least number of matches, a binary search in the result sets of the other words is performed to check if they are in the same document (for conjunctive queries), or that they are at the appropriate distance (for phrase queries).

4.6 Multiple node query

In the distributed system, the query server that a client contacts with a query, simply broadcasts RPC to all other query servers the request, and then collects and merges the results. Multiple word queries are executed on each node, so that only the intersection is sent back to the client.

5 Experimental results

5.1 Hardware and software environment

The prototype system had minimal hardware and software requirements and was designed to work on stock workstations, without strict requirements on interconnection hardware, disk subsystem, operating system, file system, or communication library. The system is adaptive to the amount of each node's physical memory.

The system runs on FreeBSD and Linux based PCs. For development we used two machines running FreeBSD 4.3; for the measurements in this section we ported the system to Linux 2.4.7 running on a cluster of 10 machines in a single room connected by a 1GB/sec Ethernet. The hardware configuration was equivalent on the nodes in each setup and very similar: dual 1GHz processors, 1GB memory, and 18GB SCSI disks.

Type	Total size(KB)	Files
WSJ 1991-2001	5,405,756	29
Web pages	9,991,347	501,317

Table 1: Sample document collections

The differences between our development and testing hardware environment were the processor brand names and more importantly the disk rotation and seek time (15k vs 10k rpm, 4ms vs 5ms). The only major problem in porting to Linux was that the `ext2` filesystem was limited to 2GB maximum filesize, and that limited some of the experiments. To avoid running in overload conditions, only one of the processors on the machines was used, the memory usage was kept under 0.5GB, and each FDIG node used only 10GB partitions on the locally attached hard drives.

5.2 Sample document collections

Two data collections were used in the presented measurements, and presented in Table 1: 5.03Gb of Wall Street Journal archives coming in a few 100–200MB text files, and a 10G web crawl starting from a few popular sites – mit.edu, yahoo.com, cnn.com.

5.3 Single node performance

The presented results here are on a subset of our WSJ collection that could be indexed on a single node. The different input size datapoints in the figures were obtained by consecutively appending the WSJ files to the dataset. Figure 2 (in the appendix) shows the generated index files total size as a function of the input size - the uncompressed index size is about 140% of the size of the uncompressed input size. Since the nodes used only 10G partitions maximum of 18 files could be present together with their index. The next limitation was because the merging was not in place – disk space of double index size was needed – that reduced the mergeable dataset to 11 files. Furthermore, the `ext2` 2GB limitation drove the number to 8 files only, and this is used for the merging graphs. These facts show how constrained a single node index generator can be; in contrast, indexing over several nodes was done without great effort.

The indexing is performed in two steps. The first step is to create a set of sub-indexes, and the second step is to merge these sub-indexes into the final local index. The amount of physical memory required by the system can be controlled by setting the maximum size of a generated sub-index.

We tested our prototype using several configurations and we show on the graphs the results of setting the inverted sub-index file size to 32MB and 512MB. The performance was quite similar for the two configurations. A large sub-index requires availability of large physical memory, however varying the sub-index size does not affect significantly the required disk space. The slight difference is only due to the space required for more internal structure in each index (each inverted index file is coupled with a corresponding lexical tree file). All of the following figures demonstrate performance and show the CPU and overall time, as well as the CPU utilization. The actual numbers here and in this and the following sections where were obtained with an unoptimized binary. The "program running time" measurement is the wall clock time to run each program on an otherwise quiescent system, and is measured with "time". The "measured by program" is obtained by "clock ()" in the main() function, and matches to the sum of user and system time measurements. Figure 3 shows the time required for indexing as a function of input data size. This function is linear and only slightly affected by the sub-index size. This again demonstrated that the performance will not be greatly improved by having more physical memory. As one can see from Figure 4, we achieved very good CPU utilization during the first step. The slightly better CPU utilization for the larger sub-index size, which results in a smaller index and thus less disk operations, confirms that the disk performance is the most significant factor. We could probably achieve even better CPU utilization with better interleaving of disk operations and processing by aggressive prefetching of input files and larger buffer caches.

Figure 5 shows the time required for the merging step. Once again our prototype demonstrates that one can achieve good performance without demanding large physical memory. In this case we found very interesting difference between two kinds of time measurement. Our interpretation is that only after the program exits from main() function, the OS flushes all dirty cache blocks to disk before finishing the process. Figure 6 shows the CPU utilization as a function of input data size. It is clear that second step (merge), unlike the first step (indexing), is much severely bounded by disk performance. Figure 3 and Figure 5 show that the system throughput is linear and easily predicted from the input data size. In the given hardware configuration without much optimizations, we achieved about 8.5 GB/hour. Given these measurements and the predicted linearity, this gives us approximately 200 GB/day.

5.4 Distributed indexing performance

The small dataset was distributed on four nodes (so that each can get a reasonable load but less than 2GB). The hashing storage distribution produced a load distribution ranging from 1.5GB to 2GB. The suboptimal balance was caused by the large size variation in the few files in the collection. The indexing time on the nodes was between 7 and 9.5 minutes, which matched the single node experiments. Merging took less than 3 minutes on all nodes, but the largest merging time was surprisingly not on the largest output set. The total indexing and merging time for the 5.4G was 12.25 minutes.

Our assumption will be that even the prototype system could process 1 TB of input data on a 10-node cluster in half a day. In our setup, however the constraint was disk space, so we were planning to confirm performance of up to 100 GB only which would take an hour. Unfortunately, we missed the time window in which we could run that experiment on the cluster, so we have results only for the small dataset. Nevertheless, the results we obtained were very promising.

5.5 Single node query performance

Our index structures predict queries to have only a small number of disk seek usually one. The experiments proved our expectation that queries will be bound by disk seeks. On the machines with advertised 4ms average seek latency, cold query time for a single matching document was 5ms on average on a small query set. After the first match queries consistently took on average 3μs. On the machines with slower disks the average cold query time was a little higher — less than 6ms on the first match, and on average 6μs thereafter.

Figure 7 shows the linear relation between buffered query results. In Figure 8, the value of "single word query time divided by matches" which starts as a disk seek time and quickly reaches the asymptote of disk throughput.

We could obtain the result for a 13 words phrase query in ten milliseconds for 1GB index. Figure 9 shows that multiple word query depends on the matches of the first word in the query queue; interestingly, conjunctive query time divided by the matches of the first word is nearly constant (0.5 microsecond) as showed in Figure 10, which is the same as the Figure 8.

6 Multiple node performance

Immediately after merging was done, queries could be issued on the system. On a four node setup the network latency was on average 0.500 ms — this is the time between a query issue and retrieval of all results (empty for this measurement). Which is certainly dominated by the time for a cold query which we showed to be a magnitude higher. However, for repeated queries, the times are worse than the single node ones, because of the network latency. For the same case on a single node, the data is in the buffer cache. Of course, a query cache mechanism could be used to provide the same functionality.

7 Conclusion

In this paper we have presented the architecture of a distributed index generator, a prototypical implementation and its performance.

Without using very optimized data structures and indexing algorithms, we were able to achieve a good indexing performance on a single node (12GB/hour). We showed that it is easy to improve the indexing time linearly by adding more hardware on several machines (thus to achieve the 1 TB goal in a few hours in a small cluster). In contrast, adding more memory to a single node does not improve performance. Other bottlenecks in a single system will also quickly limit the performance on a single node. There is no constraint for increasing the number of nodes in indexing per se, because in our implementation, there is no need for synchronization and network communication during the indexing and merging phase. We also showed that query times on a cluster of workstations is comparable to the query time on a single machine, and can be faster in some cases. It can be further optimized with the striped approach we described.

We didn't see the direct need of large processing power to justify a peer-to-peer system just for the need of current web indexing. However, when the constraint is storage, integration of the indexing system with an existing distributed storage systems like CFS will be efficient. Furthermore, when the distributed storage system is the content provider, it can easily provide an indexing service.

There is still much work to do to improve our system. We believe that the following ideas presented in section 3 are important and have to be evaluated: striped inverted index, dynamic consistent hashing using Chord, distributed queries using recursive Chord queries, storage interface on top of CFS, input compression using zlib(3) or Huffman word encoding (as CFS extension) Elias gamma coding of the index, in-place index merging and larger index file support, full system automation and incremental updates.

References

[1] Peter Bailey and David Hawking. A parallel architecture for query processing over A terabyte of text. Technical Report TR-CS-96-04, ACT, Canberra 0200 ACT, Australia, 1996.

[2] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[3] E.W. Brown, J.P. Callan, and W.B. Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB)*, pages 192–202, Santiago, Chille, September 1994.

[4] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs, 2001.

[5] David Hawking. Scalable text retrieval for large digital libraries. In *European Conference on Digital Libraries*, pages 127–145, 1997.

[6] Bing-Chao Huang and Michael A. Langston. Practical in-place merging. *Communications of the ACM*, 31(3):348–352, 1988.

[7] Timothy C. Bell Ian H. Witten, Alistair Moffat. *Managing Gigabytes: Compressing and Indexing Documents and Images*, pages 79–82. Van Nostrand Reinhold, New York, 1994.

[8] Alistair Moffat Justin Zobel and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, 1998.

[9] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[10] Jyrki Katajainen, Tomi Pasanen, and George Titan. Asymptotically efficient inplace merging. In *Mathematical Foundations of Computer Science*, pages 211–220, 1995.

[11] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications, 2001.

[12] J. Zobel, A. Moffat, and R. SacksDavis. An efficient indexing technique for full text databases. In *Proceedings of the 18th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Vancouver*, pages 352–362, 1992.

[13] J. Zobel, A. Moffat, and R. SacksDavis. Searching large lexicons for partially specified terms using compressed inverted files. In *Proceedings of the 19th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Dublin*, 1993.

[14] Justin Zobel and Alistair Moffat. Adding compression to a full-text retrieval system. *SP&E*, 25(8):891–903, 1995.

Fig 2. Output size vs input size

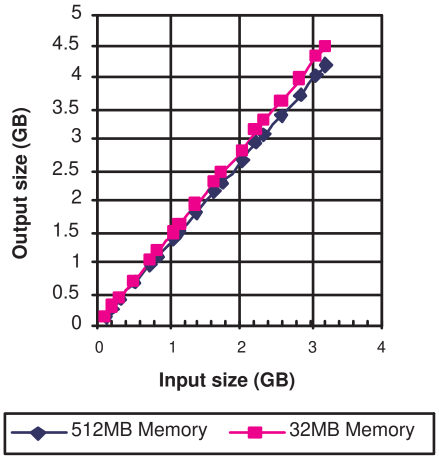

Fig 3. Indexer CPU time and running time

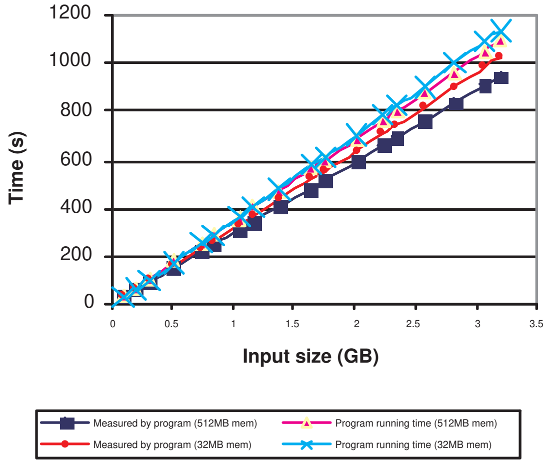

Fig 4. CPU usage for indexing

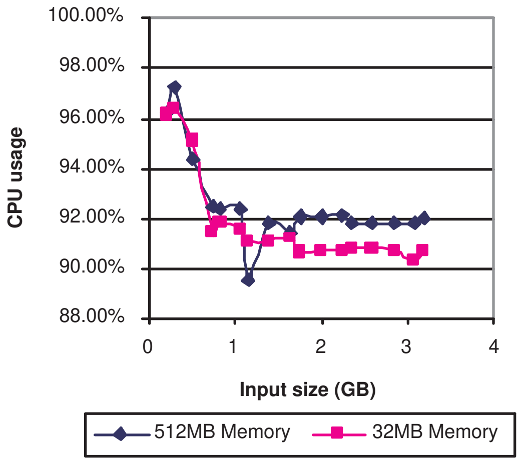

Fig 5. Merging time measured by program and program running time

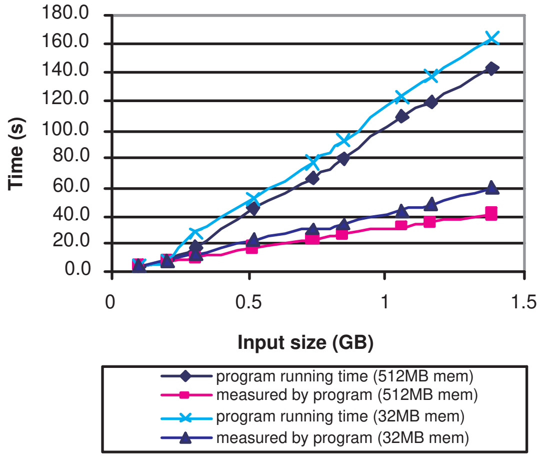

Fig 6. CPU usage for merging

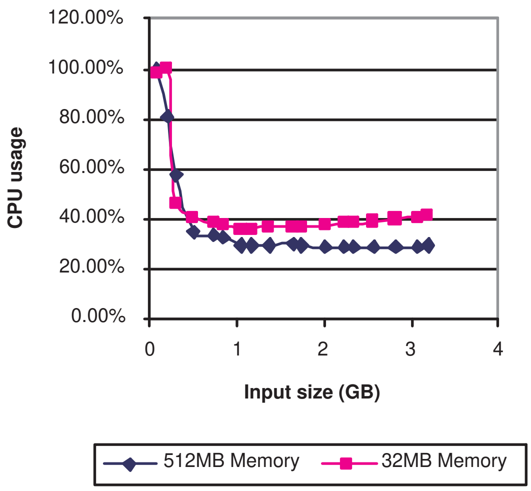

Fig 7. Single word query time (average) vs matches

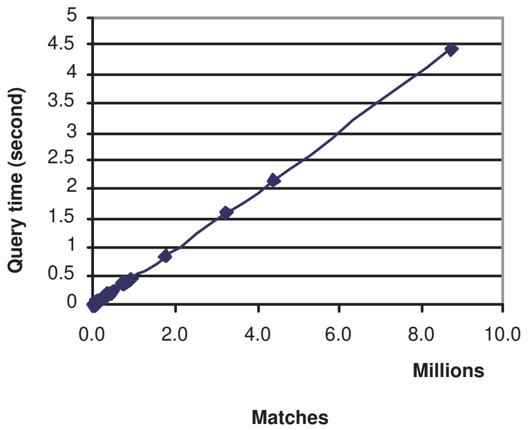

Fig 8. "Single word query time divided by matches" vs matches

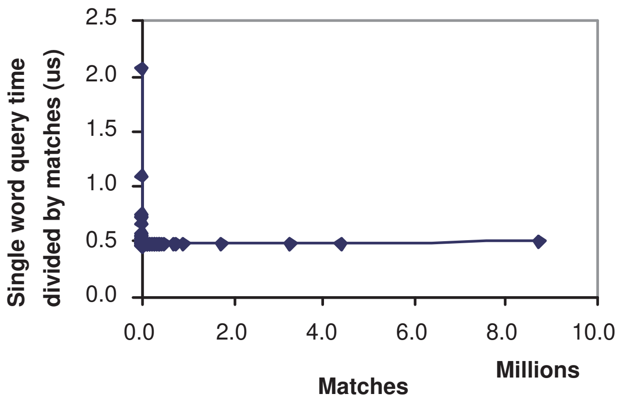

Fig 9. Query time vs the matches of first word

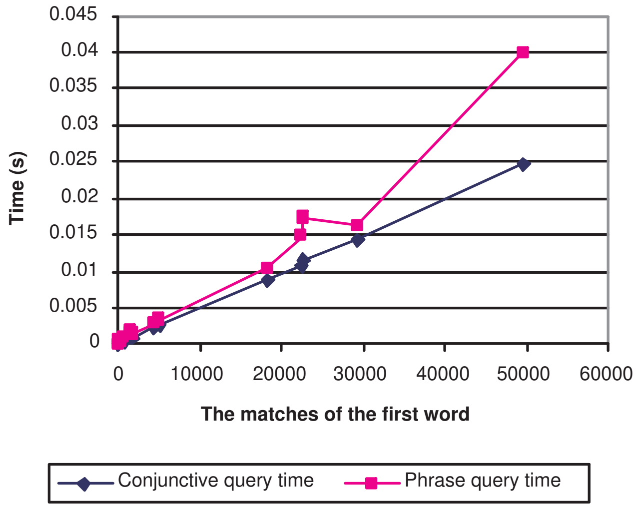

Fig 10. Query time divided by the matches of the first word vs the matches of the first word