RI 08009 June 10, 2008 Computer Science

IBM Research Report

Scalable Algorithm for Distributed In-Memory Text Indexing

Ankur Narang, Vikas Agarwal, Monu Kedia, Vijay K Garg

IBM Research Division
IBM India Research Lab
4, Block C, Institutional Area, Vasant Kunj
New Delhi - 110070. India.

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

Scalable Algorithm for Distributed In-Memory Text Indexing

Ankur Narang, Vikas Agarwal, Monu Kedia, Vijay K Garg
{annarang, avikas, monkedia, vijgarg1}@in.ibm.com
IBM India Research Laboratory, New Delhi, INDIA

Abstract

Text based search remains an important technique to retrieve data. Existing and emerging domains including the massive and rapidly changing Web, sensor networks, stream computing and others involve searching huge quantity of data coming at large rates of 10GB/s - 100GB/s. High input data rates in these applications impose real time constraints on indexing. This makes it necessary to have high indexing rates for large volumes of data. Future parallel architectures with storage class memories will pave the way for high speed in-memory text indexing, where index can be completely stored in a high capacity storage class memory. We present sequential and distributed in-memory text indexing algorithms. These algorithms leverage our novel index data structures that make text document indexing and merging process more efficient. We compare our design with the indexing algorithm in CLucene [1] and show that the asymptotic time complexity of our sequential indexing algorithm ($O(R.\gamma/k)$) is better than CLucene ($O(R.\gamma/k*(\log(R/k)+\theta))$), where R is the number of documents, γ and θ are document-term distribution parameters and k is the merge-factor parameter in index construction. We also present the asymptotic time complexity for our distributed indexing algorithm and show that it is better in scalability compared to distributed version of CLucene. The experimental results of our sequential & distributed indexing algorithm using actual website data on an MPP architecture (BG/L [4]) are also presented along with comparison to sequential & distributed indexing using CLucene. We demonstrate around $2\times$ improvement in indexing performance for the sequential algorithm and around $3\times$ - $7\times$ improvement in indexing performance for the distributed algorithm.

Keywords : Algorithms, experimentation, performance, text indexing, in-memory indexing, distributed indexing, Lucene, asymptotic time complexity

1 Introduction

Text based search remains an important technique to retrieve data including images, movies and sounds recordings. The current distributed IR systems are expected to maximize search throughput while having low acceptable response times. They typically generate document partitioned index where each index slice is generated by a single node (could be single or multi-core processor). Distributed search is performed and search results are then merged to generate the final top-X (50,100) documents for a query. Since, disk based accesses are involved in indexing, the indexing speed is limited by memory size and disk access times. Optimization is primarily focused at disk-based storage and distributed access of index and text ([13], [5], [8], [11]). However, recent trends including need for real-time indexing and search for massive amounts of data, along-with advent of massively parallel (multi-core) architectures and storage class memories [7], motivate exploration of performance optimizations for in-memory text indexing and search.

The Web is so large and growing so rapidly that the time for building index is a significant factor in providing effective search output. Also, the contents in the Web change extremely rapidly. This necessitates either efficient incremental index updates or rebuilding approach. Typically, for efficiency and simplicity,

[1] Refer http://www.sourceforge.net/projects/clucene. CLucene is a C++ implementation of Apache Lucene (http://lucene.apache.org/java)

index rebuilding approach is taken which makes it necessary to have low indexing time over huge volumes of data.

In future there will be a strong need for real-time indexing of massive amounts of data flowing at the rate of 10s - 100s of GB/s. Imagine, sensor networks sending massive amounts of data that needs to be searched for patterns and the search results are time-critical like an earthquake prediction, critical health conditions of patients, volcanic eruptions, climate warning systems and so on. Such scenarios cannot tolerate any violation of strict response times. Also, the data and index will be expected to age-off in some fixed time. Hence, we need extremely fast indexing rates that can enable delivering tight constraints of search response time in large scale systems.

Rapid advances in computer architecture and system software in recent years have produced massively parallel systems like BG/L (by IBM Research). In near future, one can expect to see massively parallel multi-core systems (1K - 2K cores) with storage class memories. For such systems, one can store the complete index and text in memory of about couple of hours of data. Thus the index data structures need to be re-designed to attain high indexing rates. One also needs to re-design indexing and search algorithms to execute efficiently on these systems.

In distributed IR environments, one way to deal with massive amounts of data and have stringent constraints on response times is to have distributed index by partitioning the total input data across groups of nodes and to construct a single index per group (instead of per node basis). This group-index is constructed by merging the indexes from each node in the group. Such a group-based construction helps in reducing the number of nodes involved in search which helps in reducing the search response time, especially in cases where global scoring requires lot of communication between the search nodes.

We study both the sequential text-indexing problem and distributed text-indexing problem for in-memory indexing and design data-structures and algorithms to provide scalable solution to both the problems. We use document partitioning instead of term-partitioning as we intend to scale our design and implementation to thousands of processors. The term-based partitioning strategy can lead to load-imbalance which could be seriously detrimental to performance in a large-scale system [13].

In this paper, we look in detail at the indexing process in text-search engines and design new data-structures and sequential algorithm as well as distributed algorithm for scalable indexing. We compare the sequential performance and also the distributed performance (on an MPP architecture) of our implementation of the scalable text indexing algorithm vs. the indexing algorithm in CLucene.

The paper makes the following contributions:

- We present the design of a novel data-structure for in-memory index (storage) that makes the whole text indexing process more efficient. Leveraging this new data-structure, the paper presents the design of sequential in-memory text indexing algorithm. We analyze the asymptotic time complexity for both cases and show that our algorithm has better asymptotic time complexity. We also compare the sequential performance of our text-indexing algorithm vs. CLucene (both of them with in-memory indexing) and show around 2x improvement in indexing performance.

- We present the design of our distributed in-memory text-indexing algorithm and compare it against distributed algorithm using CLucene data-structures and compare their asymptotic time complexities. We show that our distributed algorithm has better asymptotic time complexity. We compare the results using actual website data on an MPP platform and demonstrate around 3x-7x gain in indexing performance.

We note that our design of data-structures and algorithm for Indexing is *independent* of the architecture. So, it is really applicable to Clusters, Cluster of SMPs, large-scale SMPs and MPPs like Blue Gene and other distributed architectures.

This paper is organized as follows. Section 2 gives related work in the area of parallel and distributed text indexing. Section 3 explains the indexing process in CLucene and its performance drawbacks for in-memory indexing. Section 4 dives down deep into the design of new data-structures for indexing, optimization of these data-structures, analyzing the time complexity of sequential indexing using these new data-structures and comparing this with sequential indexing in CLucene. Section 5 goes into detailed design and analysis

of the distributed indexing algorithm. It also gives time complexity analysis for our distributed algorithm and CLucene. Section 6 presents the results of the experiments and Section 7 concludes with summary and future work.

2 Related Work

Distributed Information Retrieval has been a well-studied area for more than last two decades. In this section we provide an overview of previous efforts that provide an insight into text indexing performance apart from other IR performance issues.

Urs Holzle, Jeffrey Dean and Luis A. Barroso [1] give an overview of frameworks and techniques to handle huge volumes of data and queries. They use document partitioning for index construction and let a single query use multiple processors to achieve lower response time. The authors argue in favor of large-scale multiprocessing as compared to large shared memory machines, using the fact that the indexing/search problem has higher ratio of compute to communication time justifying a large-scale distributed approach. In contrast, we study distributed indexing scalability on large number of nodes assuming that index is stored on memory only.

Jeffrey Dean and S. Ghemawat [3] explain a generalized Map-Reduce framework. Here the programs written in functional style, are automatically parallelized and executed on a large cluster of nodes. The run-time system takes care of data-partitioning, scheduling across multiple nodes and manages inter-node communication and node failures. Using this framework they address the problem of large-scale indexing by running the indexing process as a sequence of five to ten Map-Reduce operations. Our in-memory indexing scalability techniques can be programmed in the run-time of this system, which will improve the performance of this system for large-scale indexing. Hence, our indexing scalability optimizations are complementary to the Map-Reduce framework and can be applied to any distributed IR system.

In [6], Sergey Melnik et.al. introduce software pipelining for indexing on a single node to exploit intra-node parallelism. They use an embedded database system and study mixed-lists to efficiently support zig-zag joins by using the ability to extract limited portions of the inverted-lists. In contrast, we study inter-node parallelism for indexing, assume index is stored in memory only and design innovative data-structures to make the indexing process more efficient.

Stanfill et.al. [9], evaluate a document ranking algorithm for both in-memory index database and disk-based index storage. Here, the inverted lists are distributed across available processors in a mixed term-wise and document-wise fashion which results in increase in communication and non-scalability for large number of processors as also has been indicated in the paper. We study distributed indexing on an MPP system with in-memory index but with only document-based partitioning and group-based index construction which leads to better scalability and performance in most cases.

In [2], several methodologies for Distributed IR are considered and compared on the basis of performance, efficiency and effectiveness. The methodologies are differentiated by the kind of data that must be held by the receptionist, varying from no more than a list of valid sub-collections (central nothing) to a merged vocabulary (central vocabulary), to a full-index of stored data (central index). While [2] compares multiple methodologies, for librarians/receptionists based distributed IR, we focus on sequential algorithm improvements for in-memory text indexing as well as scalability of distributed in-memory indexing in case of large scale systems.

3 Scaling Issues For In-Memory Indexing In Lucene

The Lucene index is organized in segments. Each segment contains information about terms and their frequencies, documents in which they occur and positions in which they occur in the documents. The positions and frequencies are stored in sequential fashion and accessed by adding base position with offset. The terms are also kept sorted for fast binary search during query evaluation. The whole organization is designed to minimize the number of disk accesses as the disk seek time is orders of magnitude larger compared

to compute or memory access time. The index also stores skip-lists for documents to enable fast access of documents (in O(log(n)) time).

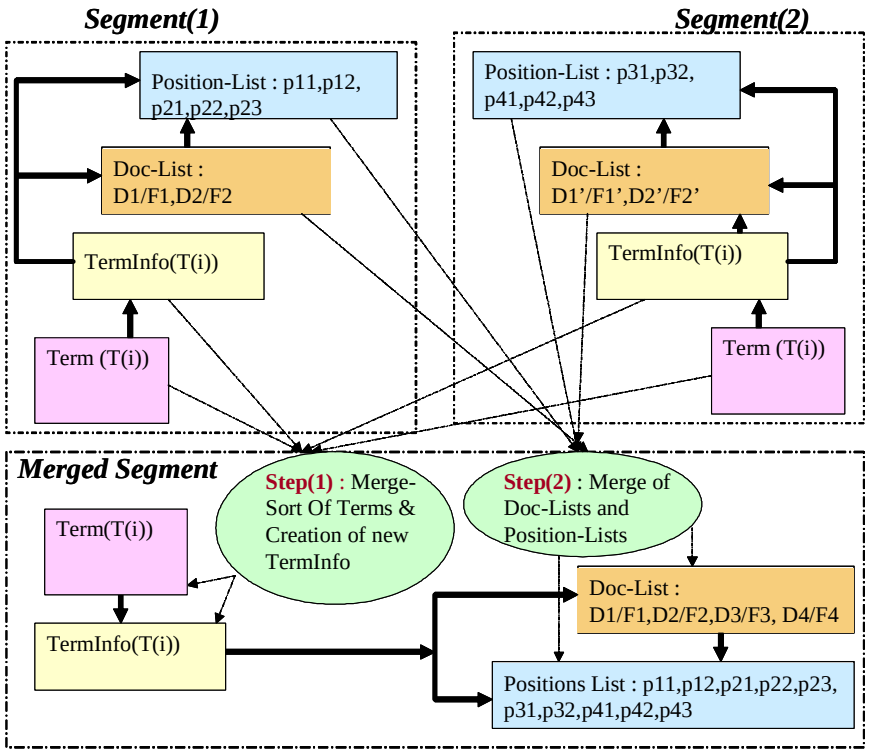


Figure 1: Lucene index merge process

During the merging of segment indexes, a merge-sort is performed to keep the terms sorted in the merged index and the document IDs are updated to reflect the new set of documents. The control structure for the final merged index is updated at each step to reflect the new merged Index. There are two key steps involved in merging of segments into a next level segment. First (Step(1) in Fig. 1), is the k-way merge sort of the sorted-term-lists in k segments to be merged. Second (Step(2) in Fig. 1), is reading the document and postings data per unique term and then copying it to the destination segment. As the number of merge increases this data is again and again read and copied over to successive merged segments. This process along with sorting of terms makes the merge process inefficient.

Hence, the current indexing algorithm in CLucene is not scalable. That is, if we double the number of processors for constructing one merged index (for the same data size), indexing may not necessarily get a speedup close to two. This is because the index-merge process becomes the bottleneck quickly.

4 Design of Data-Structures & Algorithm For In-Memory Text Indexing

In this section we first present our design of data-structures for efficient indexing including index-merging. Then, we look into optimization of this data-structure with respect to space-time trade-offs in indexing and search. Next, we compare the asymptotic time complexity of our design vs. CLucene.

We note that this paper deals with improvements in time of indexing for our approach over the current approach in CLucene, but not on space (memory) comparison. However, based on experimental measurements we are around two times the memory needed by CLucene index.

4.1 Indexing Data Structure Design

The index format of CLucene, as explained in the previous section, is not optimized for in-memory index storage and query search. Hence, we design new data structures for index storage. To eliminate the need to sort the terms we use hash-tables and terms/documents as keys in the hash-tables. To reduce the inefficiency in the merge process of repeatedly re-organizing document and postings data into higher level segments, we propose a 2-dimensional hash-table based approach. In this approach we keep a top-level hash-table called GHT (Global Hash Table), that maps unique terms in the document collection to a second-level hash table. In the second-level hash-table, called IHT (Document Interval Hash Table) the key is a range of documentIDs (document-interval) and the value is the detailed document and postings data for that term (in GHT). This avoids repeated re-organization of data in the segment indexes that get merged in the final merged index and hence makes the merge process more efficient compared to CLucene. This merge efficiency cannot be attained in a single-dimensional hash table. The details on this design and its optimizations are described below.

We use a 2-dimensional hash table structure in memory to store the inverted-index for a collection of documents. The first dimension is for terms (including Field Name) and the second is for range of documentIDs. For each term entry, there is a second level hash table which is indexed by range of documentIDs. After indexing by documentID-interval into this second-level hash table, we get the list of documents that contain that term and for each document we get the details on frequency of the term in the document and pointer to the list of positions of occurrence of the term in the document. Fig. 2 illustrates the conceptual structure of the Global Hash Table (GHT).

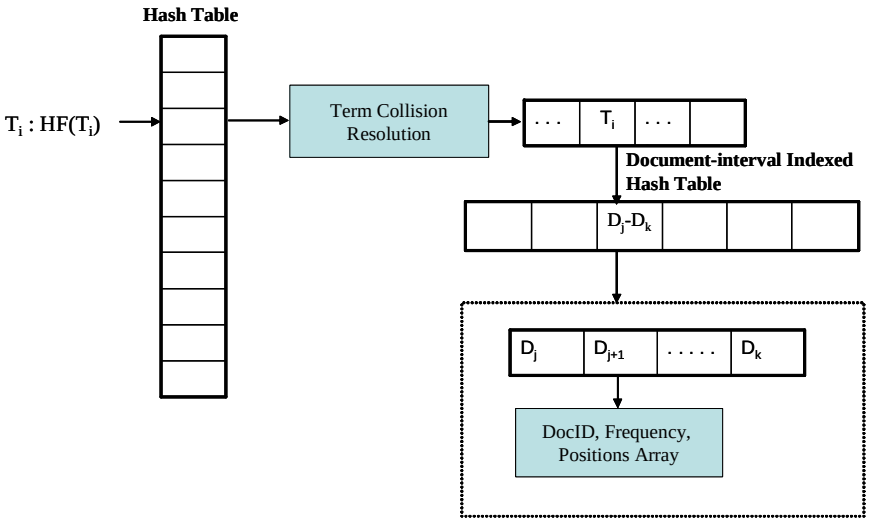


Figure 2: Global Hash Table (GHT)

Fig. 3 illustrates the structure of IHT. Here, each term in IHT points to list of documentIDs that contain that term. Each entry in this list contains the documentID, the term-frequency in that document, and pointer to the postings data for that term in the document. For IHT construction, first, the posting table for each of the k documents is formed without any need for sorting of terms. Then these posting tables are used to construct the IHT, that stores the positions for term occurrences in these k documents. The second-level hash-table, IHT, helps in scalable distributed indexing by providing the ability to offload the construction of index for a set of documents to another processor before merging that into GHT.

The IHT is then encoded into two contiguous arrays : one that allows hash-function based access into the term/document frequency data (as shown in Fig. 4) and another for actual positions data of the terms in the k documents (not shown). Fig. 4 explains the encoded IHT array structure with six sub-arrays and the steps to retrieve term positions in a document. Given, a term $T(i)$ and document $D(j)$, we perform

random-access on the sub-arrays, one-by-one, to finally get the positions of occurrence of $T(i)$ in $D(j)$. This encoding is special in that it keeps the access to the data efficient instead of sequential-traversal based. The encoding allows efficient communication from the processor that produces this IHT to the node that merges the IHT into GHT. It also helps in reducing the memory usage by enabling the application of standard index compression techniques [13]. The terms IHT and GHT refer to the above defined concepts when used in the rest of the paper.

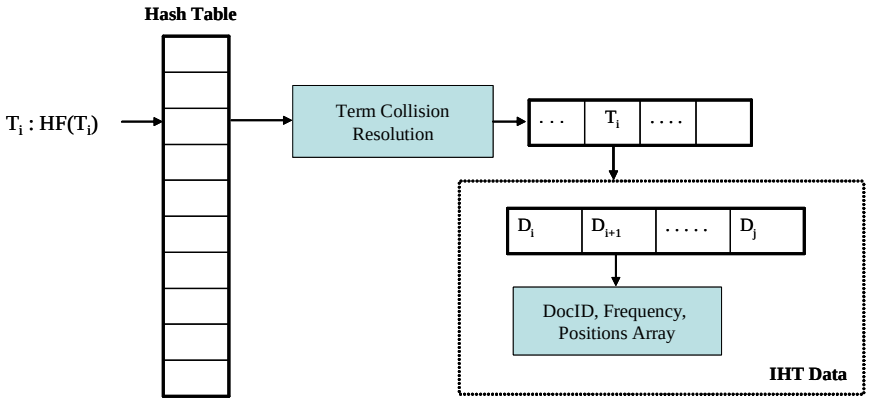


Figure 3: Interval Hash Table (IHT)

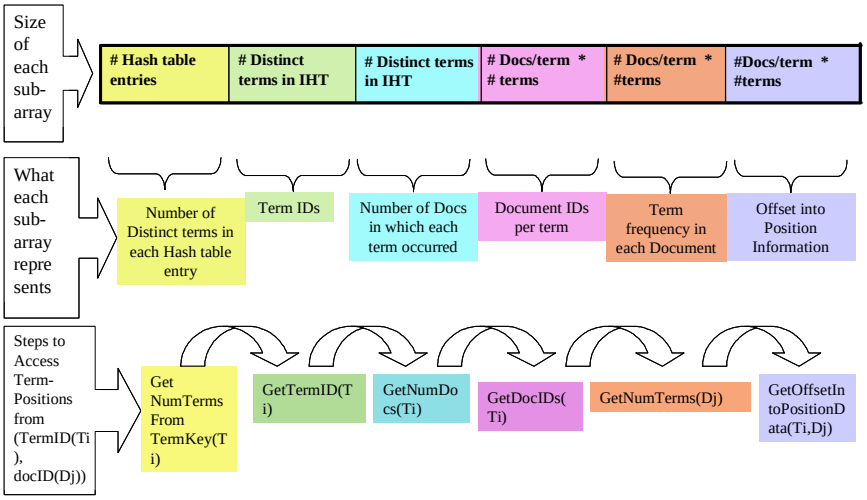


Figure 4: Encoded IHT

So, the new indexing algorithm has three main steps. First posting table (hash-table) for single document is constructed without doing any sorting of terms. Then, the posting tables of k documents are merged into an IHT, which are then encoded appropriately. Finally, the encoded IHTs are merged into a single GHT in an efficient manner. These top-level steps are similar to CLucene, but the new data-structures used (GHT/IHT as explained above) and the efficient merge process (described in sub-section 4.2) make our indexing algorithm faster and more scalable.

4.2 Optimized GHT Design & Construction

We consider the optimization of the size of the index along-with indexing time complexity (including IHT and GHT construction) and postings retrieval time during search. These are conflicting objectives as in a typical space-time optimization issues of simultaneous data-structure-size and algorithm-time-complexity optimization problem. Specifically, we consider minimizing both the GHT construction time and size of GHT while maintaining $O(1)$ time for insertion in GHT of reference to an IHT for a term, and, also $O(1)$ time for retrieval of IHT numbers given a term, from the GHT.

In the two-dimensional hash-based GHT structure, for every insertion of the pair ¡term, IHT#¿ one has to do doc-interval based hash-function evaluation and collision resolution apart from setting appropriate pointers, which takes $O(1)$ time for insertion and retrieval time. But, optimization of the size of the GHT becomes harder due to second level hash-table for document intervals and manipulation of pointers. This is supplemented by memory fragmentation by heap based allocations.

We could consider storing a bit-vector per term. Here, every bit represents an IHT and a value of '1' denotes that the term is present in some document of that IHT. But, this can lead to very high memory requirements for GHT. Hence, this approach is not taken.

Another way of optimizing the size of GHT, is to make the storage proportional to the actual number of doc-intervals per term. Typically, the [term, docInterval] matrix is very sparse. So, we take an empirical approach to optimize this index. Instead of storing the complete bitVector, we can simply store the IHT numbers that denotes those intervals that contain the term. This helps in getting significant reduction in the size of the index. This design is also better for search response time compared to traversing the bitVector (linearly or hierarchically) in the previous design.

So, the GHT contains a hash-table where the key is a unique term in the document collection and the value is a list of IHT numbers. Each IHT has at least one document that contains that term. Our index also has an array called the Array-of-IHTs whose each entry point to the IHT corresponding to that document interval. The GHT is constructed by merging IHTs one at a time into the GHT. The steps for merging an IHT into the GHT are as follows:

1. Insert pointers to the IHT data including encoded IHT data array and the positions array into the Array-Of-IHTs. This insertion happens at that entry in Array-of-IHTs which represents the document-interval corresponding to the current IHT being read. In Fig. 5, the entry g points to $IHT_{(g)}$.

2. The unique term list in the IHT is traversed. For each term, position of that term is identified in the GHT using hash-function evaluation and term collision resolution. Then, in the IHT-list for that term, the current IHT number is inserted. Fig. 5 illustrates construction of GHT from IHT using this mechanism. It shows how $IHT_{(g)}$, is merged into the GHT. First (Step-*S1* in Fig. 5), $IHT_{(g)}$ is pointed to by the appropriate location in the Array-of-IHTs. Then (Steps-*S2(a) & S2(b)* in Fig. 5), $IHT_{(g)}$, is inserted into both IHT-lists corresponding to the terms T_i and T_j in the GHT.

The above merge process does not involve re-organizing of the IHT data while merging it into GHT compared to CLucene which re-organizes the segment data when merging it into the final merged segment. This makes GHT/IHT design efficient for distributed indexing.

4.3 Asymptotic Time Complexity Comparison of S_{ghtl} vs. S_{orgl}

We refer to original sequential CLucene code as S_{orgl}. The new sequential in-memory text indexing algorithm that we designed and implemented is referred to as S_{ghtl} in the paper.

We compare the asymptotic time complexity of S_{ghtl} vs. S_{orgl} and show that S_{ghtl} has better asymptotic time complexity for indexing. The time complexity is dependent on certain term-document distribution parameters. These are mentioned below (same notation used for rest of the paper):

- α : Number of unique-terms-per-document (averaged over all documents).
- β : Number of term-occurrences-per-doc-per-term (averaged over all documents and terms).

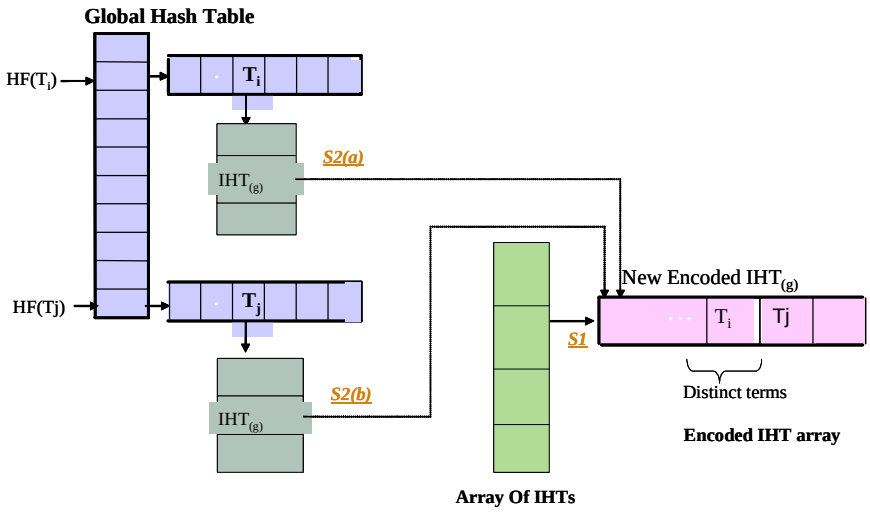


Figure 5: GHT construction from IHT

- $\gamma(k)$: Number of unique-terms in a set of k documents (averaged over all document sets of size k). This will be referenced as γ in the paper, unless qualified with value of parameter k (doc-interval size).
- θ : Number of docs-per-term in a set of k documents (averaged over all document intervals of size k and over all terms in the interval).
- δ : Number of unique-terms in a set of documents that represents one complete index.

We note that $\alpha = \gamma(1)$, $\delta = \gamma(R)$ where R is the total number of documents in the index. β is dependent on the intra-document term occurrence while, θ represents the inter-document occurrence of the same term in a given collection of documents. Both γ and θ vary with k, number of documents considered in a set. γ increases at a fast rate with increase in k but is always less than $\alpha * k$, while θ grows very slowly with k. Since, α and β are per-document parameters, averaged appropriately, they are treated as constants in the paper. These relationships amongst the parameters further clarify their significance to performance analysis in the coming sections.

There are the three phases in text indexing which we compare between S_{orgl} and S_{ghtl}:

- Single document index creation in a simple hash table (referred to as LHT).
- Single document index merging & creation of first-level multi-document index (segment / IHT).
- Merging of multi-document indexes & creation of final merged index.

We compare the time complexity in each phase and build the expressions for overall time complexity for both S_{orgl} and S_{ghtl}. In S_{orgl}, when segments are created from individual documents the terms are kept in sorted order. The sort operation takes time proportional to $[\alpha * \log(\alpha)]$.

In S_{ghtl}, the terms are kept organized in a hash-table structure, so the hash-computation is all that is needed to insert the term at its right position in the LHT. Hence, the analytical complexity is proportional only to α.

Now, assume for merging that we have only two levels of merging. This means that first single document segments are merged into a larger segment of k-documents. Then, these k-doc-segments are merged into one final segment. We compare the times for each of these separately and show how S_{ghtl} is better than S_{orgl} in both cases.

During, **first level of merge**, when k-doc-segment is formed from 1-doc-segment, then S_{orgl}, does a merge sort over the sorted-term lists of k 1-doc-segments. This work is proportional to $[k * \alpha * \log(k)]$.

Also, the data for each term is taken and copied over to the k-doc-segment. This includes postings for each document that has that term. Here, the work done is proportional to (number of unique terms in a document * #postings/term/doc * #documents/Segment). Thus, the total work is proportional to $[(\alpha * \beta * k) + (k * \alpha * \log(k))]$.

In contrast, for S_{ghtl}, no merge sort is needed, as the final unique list of terms in an IHT (equivalent to k-doc-segment of S_{orgl}) is also kept in a hash-table. The work involving copying term-postings per document and term is roughly the same. Hence, for S_{ghtl}, the work is proportional to $[(\alpha * k * \beta) + \gamma]$. The following equation represents the IHT production time.

$$T(IHTproduction) = O((\alpha.k.\beta) + \gamma) \tag{4.1}$$

Now, in the **second level of merge**, S_{orgl}, takes similar approach as the first level of merge. That is for each term-posting per document and term, it performs copying operations and updating offsets for disk-based storage. This work is proportional to $[\gamma * \theta * \beta]$ per k-doc-segment. Also, it performs $k'(= R/k)$-way merge of k-doc-segments into a final segment. If we consider final segment to consist of $(k * k')$, documents, then the merge sort effort is proportional to $k' * \gamma * \log(k')$. Thus, the total effort by S_{orgl} in the second level is $[k' * \gamma * \theta * \beta] + [k' * \gamma * \log(k')]$.

In contrast, for S_{ghtl}, we do not need to perform work proportional to number of documents in an interval (i.e. k). Here, we simply take the constructed IHT and insert a pointer to it in the Array-of-IHTs. This is a constant time operation. Also, we traverse each term in the IHT and set the IHT number in the GHT. This work is proportional to number of unique-terms in an IHT i.e. γ. The hash table insert operations are proportional to δ. Total work for the second level merge for S_{ghtl} is proportional to $[\ (\gamma * k') + k' + \delta]$, and is given by the following equation:

$$T(IHT\,merge) = O(\gamma * R/k + \delta) \tag{4.2}$$

So, at each of the three phases, S_{ghtl} is better than S_{orgl}. Adding, the times for all the three steps we get the asymptotic complexities, as,

$$\begin{aligned} T(S_{orgl}) = O((k.k'.\alpha.\log(\alpha))+ \\ k' * (\alpha.k.\beta + k.\alpha.\log(k))+ \\ k' * (\gamma.\theta.\beta + \gamma.\log(k')) \end{aligned} \tag{4.3}$$

Simplifying, and using $R = k * k'$, we get,

$$T(S_{orgl}) = O(R.\alpha.(\log(\alpha.k) + \beta) + R/k * \gamma.(\log(R/k) + \theta)) \tag{4.4}$$

Assuming α, β as constants and simplifying, we get,

$$T(S_{orgl}) = O(R/k * \gamma.(\log(R/k) + \theta)) \tag{4.5}$$

Now, for S_{ghtl} we have,

$$T(S_{ghtl}) = O((k'.k.\alpha) + k'.(\alpha.k.\beta + k.\alpha) + k'.\gamma + \delta) \tag{4.6}$$

Simplifying, we get,

$$T(S_{ghtl}) = O(R.\alpha.\beta + (R/k * \gamma)) \tag{4.7}$$

Assuming α, β as constants and simplifying, we get,

$$T(S_{ghtl}) = O(R/k * \gamma) \tag{4.8}$$

Comparing, equation (4.8) with (4.5), one can see that S_{ghtl} has better asymptotic time complexity than S_{orgl}. The actual run-time of S_{ghtl} is also better compared with S_{orgl} as shown with experiments in section 6.

5 Distributed Algorithm For Text Indexing

In this section we explain the distributed indexing algorithm. The distributed algorithm using the CLucene indexing and merging code is referred to as P_{orgl} while the distributed algorithm using our indexing algorithm is referred to as P_{ghtl}.

5.1 Distributed Indexing Algorithm Design

The nodes in the distributed system are partitioned into index-groups. Each index group of size $(P+1)$, has P Producer nodes and one Consumer node (also called index-group head). Total text data is partitioned document-wise and assigned to the index-groups. Within each index-group the data is divided equally amongst the Producer nodes. Hence, instead of creating one index for the complete document set or one index per node in the system, we actually create one index per index-group which results in distributed index for complete set of documents in the system.

In case of P_{orgl}, each Producer generates segments (one segment represents index for limited, k, number of documents) for the data provided to it. These are then sent to the Consumer that merges them into the final merged segment using P-way merge procedure.

In case of P_{ghtl}, the Producers, construct IHTs (each IHT represents one inverted index for a set of documents). All Producers store the IHTs locally and then send the IHT-meta-data to the Consumer. Here, IHT-meta-data refers to terms and the document frequency per term in the IHT. This helps in saving space at the Consumer node. The Consumer takes each IHT-meta-data and merges it into a single GHT maintained by it. The distributed indexing and search algorithm is illustrated in Fig. 6 below.

For search, same query is provided to each Consumer(index-group head) that has the merged index. Each Consumer determines the matching documents for that query. The Consumers share matching document information for global scoring. Then, each Consumer scores the matching documents and selects top-N documents followed by across-Consumer top-N selection and reporting to the user.

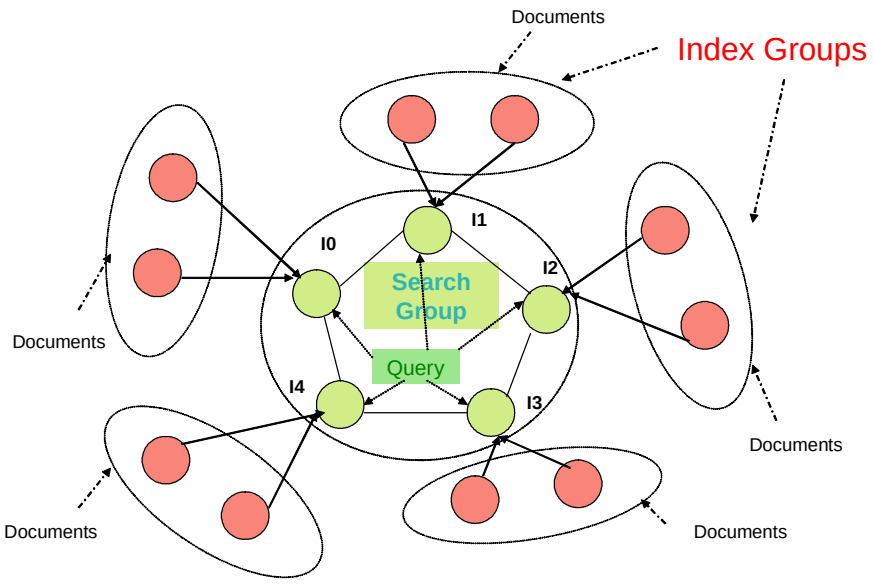


Figure 6: Distributed Indexing & Search

The pipeline diagram in Fig. 7 illustrates the steps involved in distributed indexing. These overall steps are applicable in both cases - P_{ghtl} and P_{orgl}. These are:

- Generation of encoded IHTs(P_{ghtl})/lucene segments (P_{orgl}), in parallel, at the Producers.

- Communication of IHT-meta-data/segments from the Producers to the Consumer.
- Merge of IHTs/segments at the Consumer.

These steps get repeated across a number of rounds till all the documents get indexed. In case of limited memory in the system, the documents are assumed to be aged-off after some duration. When all the documents in an IHT have been aged-off it is deleted from the index.

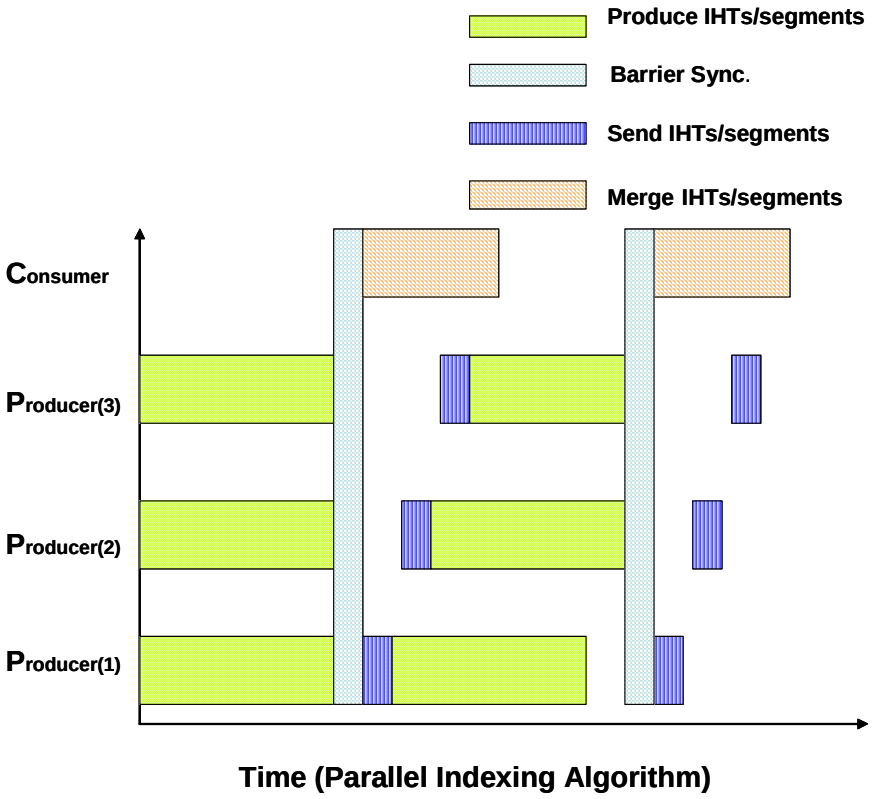


Figure 7: Distributed indexing pipeline diagram

5.2 Time Complexity Analysis of Distributed Indexing

Let, the size of the indexing group be $(P+1)$: P Producers and 1 Consumer. We can use the pipeline diagram in Fig. 7 to analyze the time complexity of both P_{ghtl} and P_{orgl}. Let, there be n rounds with P Producers in each round. Let, $Prod_{(j,i)}$ denote the total time for j^{th} Producer, in i^{th} round, where $1 \leq j \leq P, 1 \leq i \leq n$. This includes both the compute time(denoted by $ProdComp_{(j,i)}$) and the communication time for the j^{th} Producer (denoted by $ProdComm_{(j,i)}$). Similarly, $Cons_{(i)}$ denotes the total time spent by the Consumer in the i^{th} round which includes both the compute time for merging (denoted by $ConsComp_i$) and its communication time (denoted by $ConsComm_i$) in i^{th} round. The distributed indexing time is approximately given by the following equation :

$$T(distributed) = X + Y + Z \tag{5.1}$$

$$where, X = \max_{j} ProdComp_{(j,1)} \tag{5.2a}$$

$$Y = \sum_{2 \le i \le n} \max(\max_{j} Prod_{(j,i)}\ ,\ Cons_{(i-1)}) \tag{5.2b}$$

$$Z = Cons_{(n)} \tag{5.2c}$$

5.2.1 Time Complexity for P_{ghtl}

Below, we consider in detail the analysis for P_{ghtl}. Since, each Producer has to produce only $1/P$ of the number of IHTs in sequential case, its compute time gets reduced by a factor of P. Equation (4.1), gives compute time for the j^{th} Producer in the i^{th} round. So,

$$ProdComp_{(j,i)} = \{k.\alpha.\beta + \gamma\} \tag{5.3}$$

The total number of rounds,n, is given by $R/k * 1/P$, since total number of IHTs to generate is R/k and in each round P IHTs are generated.

The Consumer receives IHT-meta-data from all Producers in a round (assuming load-balance across Producers). Hence, an average Producer will wait for half of the IHTs to be sent to the consumer before it can send its own. Thus, the communication time per Producer in a round is the sum of the synchronization time(MPI collective), waiting time before its turn can occur to send the IHT-meta-data (referred as $WaitTime$) and the actual time for sending the IHT-meta-data to the Consumer (referred as $IHTSendTime$).

So, the communication time per producer, in the i^{th} round is:

$$ProdComm_{(j,i)} = \{gc(P) + IHTSendTime + WaitTime\} \tag{5.4}$$

$$IHTSendTime = \{IHTMetaDataSize/BW + Latency(P)\} \tag{5.5a}$$

$$WaitTime = P/2 * IHTSendTime \tag{5.5b}$$

where,

$$gc(P) = Sync.\,time\,(MPIcollective) \tag{5.5c}$$

$$IHTSize = Size\,of\,the\,IHTMetaData\,in\,bytes \tag{5.5d}$$

$$BW = Bandwidth\,of\,the\,network\,on\,which\,IHTmetadata\,is\,transmitted \tag{5.5e}$$

$$Latency(P) = Latency\,of\,communication\,from\,Producer\,to\,Consumer \tag{5.5f}$$

For the Consumer, we again consider the compute and communication time separately. The merge time for the consumer remains the same as it has to merge the same number of IHTs into the GHT. Using, equation (4.2), and considering merge of P IHTs per round, we get,

$$ConsComp_{(i)} = O(P * \gamma + \delta) \tag{5.6}$$

For each round of IHT transfer, the Consumer collects all IHT-meta-data available in that round before merging them into the GHT. So, the Consumer communication time is the sum of the synchronization time (MPI collective) and the time for receiving each IHT. The communication time of the Consumer in the i^{th} round is given by the following:

$$ConsComm_{(i)} = gc(P)\ +\ P * \{IHTSize/BW + Latency(P)\} \tag{5.7}$$

Total Time for the Consumer in the i^{th} round is:

$$Cons_{(i)} = ConsComp_{(i)} + ConsComm_{(i)} \tag{5.8}$$

Using equations (5.1) and (5.3) to (5.8) we can compute the analytical expression for time complexity of P_{ghtl}. The communication time in our experiments was very small compared to the overall distributed indexing time, hence we ignore the communication time in further analysis. Due to the pipelining of the produce and merge phases, the overall indexing time depends on whether the produce phase dominates the merge phase or vice-versa. Hence, we consider two cases depending upon whether IHT production compute time per round dominates merge compute time per round or vice-versa. Simplifying the expressions for each case we get the following:

Case(1) : Production time per round > Merge time per round

$$T(P_{ghtl}) = k.\alpha.\beta + \gamma + (R/kP - 1) * (k.\alpha.\beta + \gamma) + (P.\gamma + \delta) \tag{5.9}$$

Since, the last round Consumer time is dominated by the Producer time, we get,

$$T(P_{ghtl}) = O(R.\alpha.\beta/P) = O(R/P) \tag{5.10}$$

Case(2) : Merge time per round > production time per round

$$T(P_{ghtl}) = k.\alpha.\beta + \gamma + (R/kP - 1) * (P.\gamma + \delta) + (P.\gamma + \delta) \tag{5.11}$$

Simplifying, we get,

$$T(P_{ghtl}) = R/kP * (P.\gamma + \delta) \tag{5.12}$$

$$T(P_{ghtl}) = O(R.\gamma/k + R.\delta/k.P) \tag{5.13}$$

Thus, we see that the scalability of P_{ghtl} is fine in Case(1) but when Case(2) occurs then the merge time becomes the bottleneck. If we use distributed merge, then we can make the merge time also scalable with P by adaptively balancing the load across producers and the consumer. This is left for future work.

5.2.2 Time Complexity of P_{orgl}

Following similar analysis as above and using (5.1), the total time for P_{orgl} can be computed. We can substitute the time complexity expression for Producer compute and communication time and Consumer compute and communication time (section 4.3) and assume that communication time is small. Here it assumes

Next, we consider two cases, Case(1) : Production time per round > Merge-time per round

$$\begin{aligned} T(P_{orgl}) &= O(R/P * \alpha.\log(\alpha.k) + P.\gamma.(\log(P) + \theta) \\ &= O(R/P * \log(k) + P.\gamma.(\log(P) + \theta) \end{aligned} \tag{5.14}$$

Case(2) : Merge-time per round > Production time per round

$$T(P_{orgl}) = O(R/k.P * (P.\gamma.(\log(P) + \theta)) \quad = O(R/k * (\log(P) + \theta)) \tag{5.15}$$

Looking at above equations (5.14) & (5.15), we can see the scalability issues with P_{orgl}. We can use distributed merge but it will help to a limited extent. Comparing equations (5.10) & (5.13) with (5.14) & (5.15), we can see that P_{ghtl} is guaranteed to show better scalability and indexing time compared to P_{orgl}.

6 Results and Analysis

We compared both the sequential and distributed indexing performance between our implementation and CLucene (version 0.9.20). We implemented our GHT/IHT based indexing data-structures and algorithm on the original CLucene code base. For CLucene, we maintain the index in memory using the *RAMDirectory*.

6.1 Sequential Indexing Performance

In the sequential case, we used an Intel machine with 3.6 GHz Xeon processor and 4GB memory, and compared the indexing performance between S_{ghtl} and S_{orgl} using real website data. Fig. 8 shows the time for indexing text-data from 500MB - 2GB. As we can see S_{ghtl} is around 2× faster than S_{orgl}. We performed search using real queries on 512 MB text-data and found that the CLucene search time, 10.3s, is better than our search time, 12.9s. We have not yet tuned the search code to leverage our data-structures (hash table based access instead of sequential access), which we believe can deliver same or better search time. We also found that our serialized index size, around 200 MB, is within two times the index size of CLucene, around 106 MB. We can further improve our index size using index compression techniques ([10], [12]).

Time (in seconds)	Data Size			
	500MB	1 GB	1.5 GB	2 GB
Sorgl	236.80	505.01	798.56	1067.29
Sghtl	115.52	242.31	408.62	573.79

Figure 8: Sequential Indexing Performance

6.2 Distributed Indexing Performance

We studied the scalability of P_{orgl} and P_{ghtl} by doing experiments on real website data. The text data was extracted from html files and loaded into the memory of the producer nodes before the indexing time measurement is started. The amount of text-data sent to each Producer node is roughly load-balanced. For CLucene, we used that value of k so that only one segment is created from all the text data fed to a Producer. This is so as to get the best indexing performance from CLucene. If we generate more than one segment at the Producer(s) then the merge process at the Consumer takes more time.

The experiments were conducted on the BG/L [4] platform. BG/L is an MPP system developed by IBM Research. It has thousands of processor nodes (PPC 440) connected in a high bandwidth 3D torus network. We ran the experiments in co-processor mode for each node. We chose BG/L as it allows us to study indexing scalability across thousands of nodes and because it was more readily available to us than large clusters. It also gives an opportunity to explore the value of BG type architectures for large scale text-indexing and search. We note that our data-structure and algorithm design are ***independent*** of the underlying architecture.

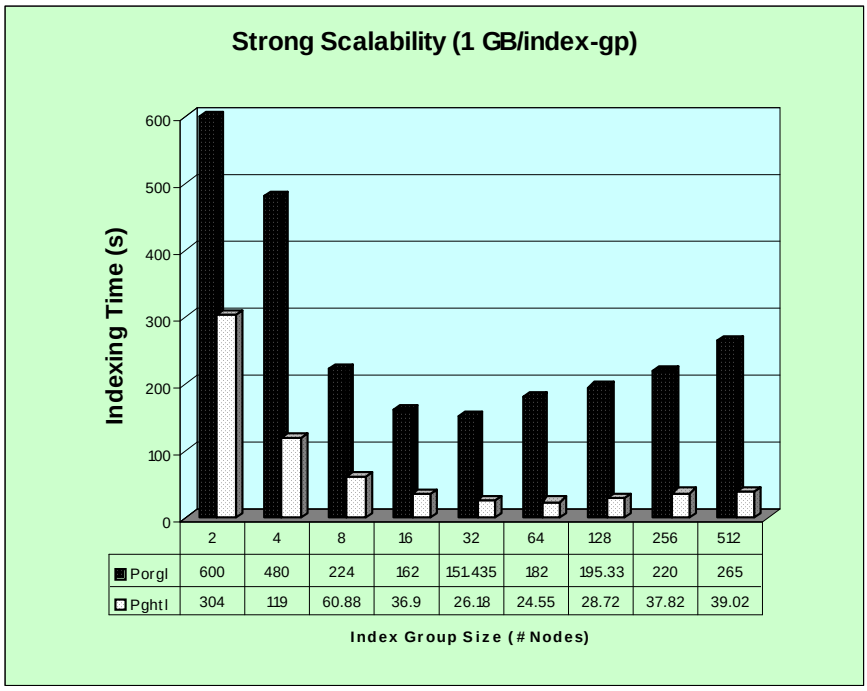
We implemented the distributed algorithm in CLucene code (P_{orgl}) as well as in our indexing code (P_{ghtl}) and compared the following aspects of indexing algorithm scalability:

- Strong Scalability : (increase P, R constant) ⇒ Indexing Time
- Weak Scalability : (increase P, increase R) ⇒ Indexing Time
- Scalability with number of docs : (P constant, increase R) ⇒ Indexing Time

6.2.1 Strong Scalability Study

In the strong scalability experiment, the input data size for an index group remains constant while the size of the group, in number of processors, is increased. We study the distributed indexing time and speedup in this setup. We determine the maximum number of processors until which the distributed indexing time keeps reducing with increase in number of processors. Below we consider index group size of G processors with P Producers and 1 Consumer, with group size varying from 2 to 512 nodes. We consider 1 GB of text-data to be indexed per group. In these experiments we measured performance by indexing large volumes of data upto 256 GB, by having multiple index-groups. Note that speedup in this section refers to increase in distributed indexing time relative to time for $G = 2$ (instead of $G = 1$).

The plots of strong scalability study for P_{orgl} and P_{ghtl} are given in Fig. 9. As we can see P_{orgl} time decreases initially from 600 s, when G=2, to 151 s, when G=32, but after this it keeps increasing for G ¿= 64. For P_{ghtl}, the distributed indexing time decreases continuously from 304 s, when G=2, to 24.55 s, when G=64, but after this it keeps increasing for G ¿= 128. Thus, both follow a U-shaped curve, but, P_{ghtl} scales till G = 64 while P_{orgl} scales only till G = 32. Fig. 10 shows the variation of speedup as the number of processors is increased. The maximum speedup obtained for P_{ghtl} is approximately 12.38, which is 3.12 times better compared to P_{orgl}, speedup = 3.9. In terms of best distributed indexing time (over all G), P_{ghtl} is approximately 6.17 times better than P_{orgl}.



	2	4	8	16	32	64	128	256	512
Porgl	600	480	224	162	151.435	182	195.33	220	265
Pghtl	304	119	60.88	36.9	26.18	24.55	28.72	37.82	39.02

Figure 9: Strong Scalability

This behavior can be explained by the inefficient merge process of P_{orgl}. For a given number of processors, P_{orgl} takes more time in both segment generation and merging compared to IHT generation and merging by P_{ghtl}. Hence, the indexing time for P_{ghtl} is lower than P_{orgl} for the same index-group size and the maximum speedup for P_{orgl}, 3.97, is lower than maximum speedup for P_{ghtl}, 12.38. Now, as the number of processors increases with the same total input text data size, the amount of text data per processor goes down and hence the segment/IHT production time goes down while the merge time increases. So, initially in each round, the production time is more than the merge time and it dominates the overall distributed indexing time. This corresponds to Case(1) in section 5.2.1 and equation (5.10) for P_{ghtl}, and similarly for P_{orgl}, this corresponds to Case(1) in section 5.2.2 and equation (5.14). However, as the index-group size increases, the

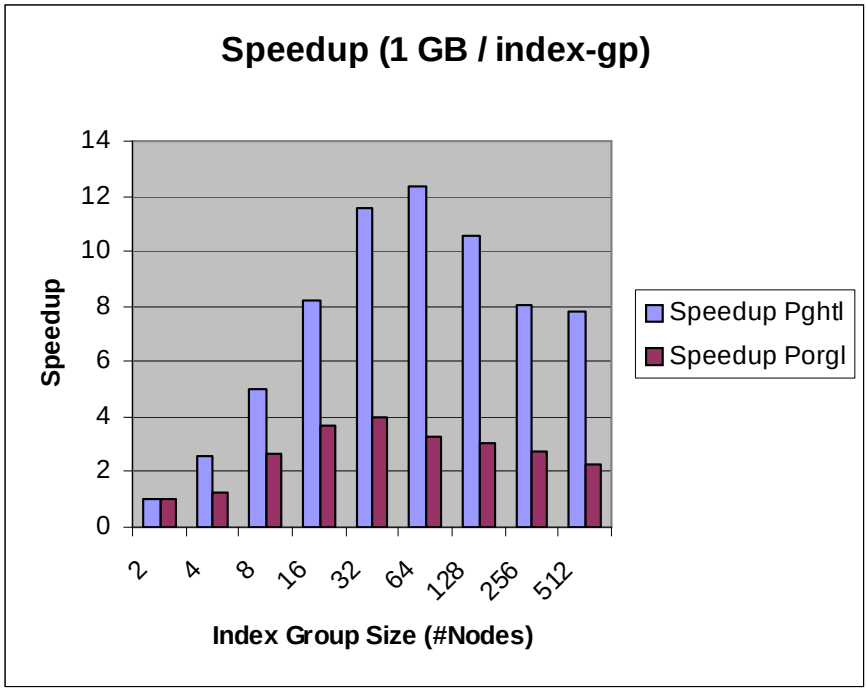


Figure 10: Speedup (strong scalability)

segment/IHT production time goes down and merge time becomes more than production time. Then the merge time dominates the overall distributed indexing time. This corresponds to Case(2) in section 5.2.1 and equation (5.13) for P_{ghtl}, and, similarly for P_{orgl}, this corresponds to Case(2) in section 5.2.2 and equation (5.15). Since, merge is performed by a single consumer node it becomes a bottleneck for scalability. Because, P_{orgl} has much more in-efficient merge compared to P_{ghtl} it reaches this bottleneck point earlier at $G = 32$, than P_{ghtl} at $G = 64$. One could also parallelize merge to shift load from consumer to producers and get better merge time. This can be done for both P_{orgl} and P_{ghtl} and hence both will gain from a distributed merge.

6.2.2 Weak Scalability

In the weak scalability experiment, both the data and number of processors are increased, and we study the increase in overall distributed indexing time. In this experiment the data to be indexed per index group is increased from 50MB to 1.6 GB, as G is increased from 4 to 128 processors. As illustrated in Fig. 11, as G increases from 4 to 128, P_{ghtl} time increases from 6.64s to 37.92s, i.e. by 5.71× factor, while P_{orgl} time increases from 15s to 290s, i.e. by 19.4× factor. This behavior can again be explained by the in-efficient merge for P_{orgl} and its dominant impact on overall distributed indexing time. So, we improve the weak scalability of distributed text indexing compared to CLucene by employing our more efficient algorithm P_{ghtl}. Note that for P_{orgl} the indexing time initially goes down from approximately 15s to 14s when G goes from 4 to 8 and data per group increases from 50MB to 100MB. This is because, in both cases the Producer compute time dominates the Consumer merge time and in the second case we have lesser data per Producer which means the Producer compute time for G=8 is lesser than the same for G=4. Hence, the overall indexing time for G=8 is less than G=4.

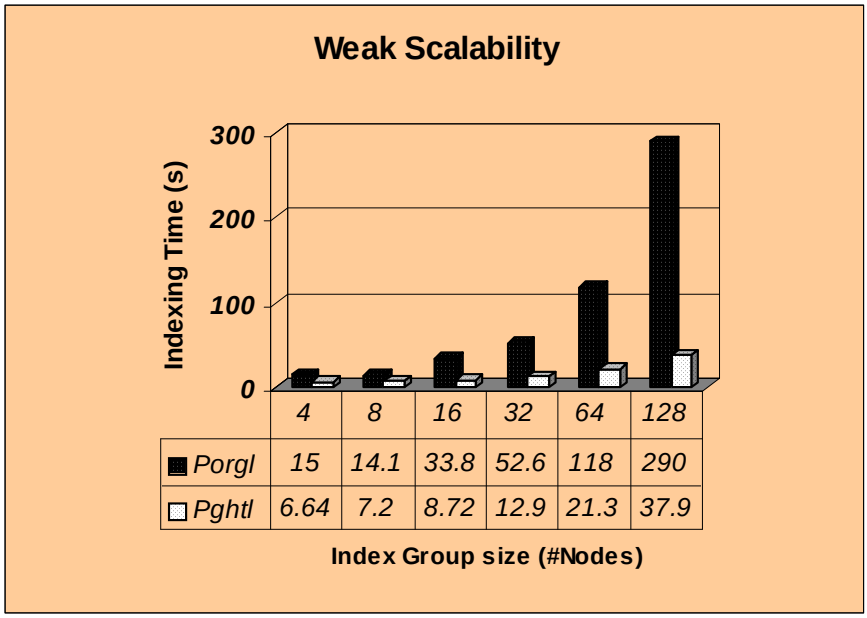


Figure 11: Weak Scalability Study

6.2.3 Scalability with increase in data size

In this experiment we study the indexing time variation with increase in text data size. Here the size of text data used to generate index in one index group is varied from 64MB to 1GB. We study the change in indexing time for both P_{orgl} and P_{ghtl} for two index-group sizes - 32 and 128. As we can see in Fig. 12 , for G=128, as the size of the data increases from 64 MB per index-group to 1 GB per index-group (16x increase), the indexing time for P_{orgl} increases from 20.23s to 191.59s (9.47 times increase), while for P_{ghtl} the indexing time increases from 3.66s to 27.97s (7.64 times increase). For G=32 (as shown in Fig. 13) the increase in indexing time is more for the same increase in text-data size. Here, a 16 times increase in data for P_{orgl} leads to 15 times increase in time while for P_{ghtl} the indexing time increases only 8 times. For both cases, G = 32 and G = 128, the merge time dominates the segment/IHT production time. For P_{orgl}, the increase of merge time with data-size is less at $G = 128$ than with G = 32. In case of P_{ghtl}, with increase in data-size there is better load-balance between the Producer and the Consumer, for the same increase in data-size it shows less increase in indexing time compared to P_{orgl} for both cases G = 128 and G = 32.

7 Conclusions and Future Work

In this paper we presented new data-structures and algorithm for sequential and distributed indexing. We have shown that our algorithm has better asymptotic time complexity in both sequential and distributed case with small increase in memory size as compared to CLucene. We supplement the claim with experiments on real data and demonstrate around 2x improvement for sequential indexing and around 3x-7x improvement for distributed indexing.

If we parallelize the merge process then we can avoid merge becoming a bottleneck. This basically involves shifting some part of the load for merge node onto to the producer-nodes to obtain load-balance between the producers and the consumer. The amount of load transferred needs to determined dynamically so that the overall indexing time can scale well with the increase in the number of total processors in an index group. As part of future work, we plan to study dynamic load balancing algorithm, and also look into index size optimization and distributed search performance analysis and optimization.

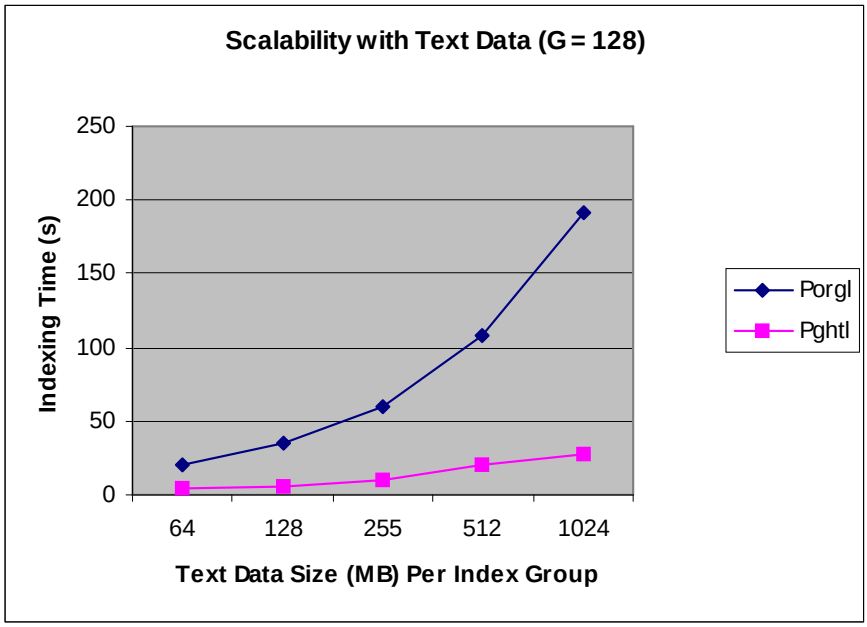


Figure 12: Scalability with increasing text-data (G = 128)

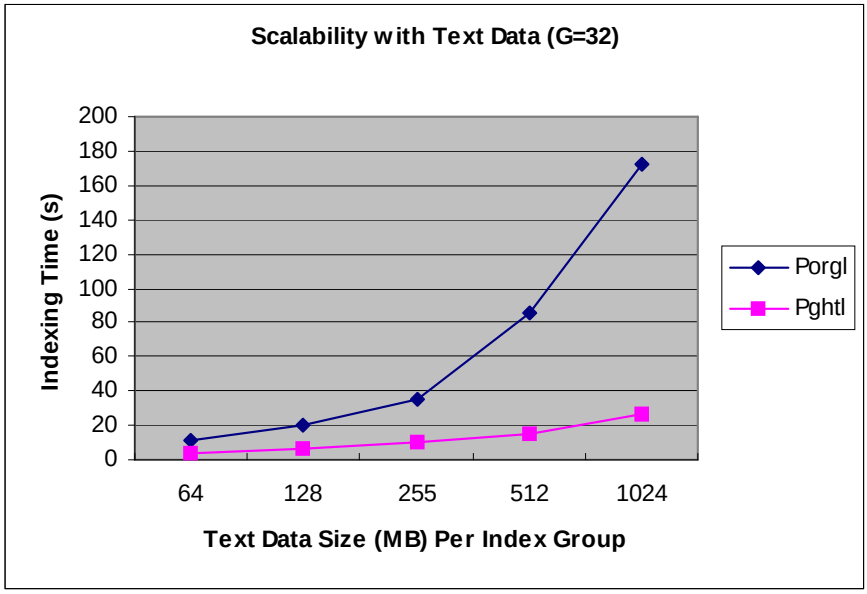


Figure 13: Scalability with increasing text-data (G = 32)

8 Acknowledgments

We thank Dr. Ravi Kothari for his strong support for this work and valuable technical suggestions and feedback all throughout. We also thank Maged Michael for his valuable feedback and suggestions. We thank all the reviewers for their thorough reading and insightful comments.

References

[1] L. A. Barroso, J. Dean, and U. Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.

[2] O. de Kretser, A. Moffat, T. Shimmin, and J. Zobel. Methodologies for Distributed Information Retrieval. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, Amsterdam, Netherlands, May 1998.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Opearting Systems Design & Implementation*, San Francisco, CA, December 2004.

[4] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, and et.al. Overview of the Blue Gene/L System Architecture. *IBM Journal of Research and Development*, 49(2/3):195–202, 2005.

[5] B.-S. Jeong and E. Omiecinski. Inverted File Partitioning Schemes in Multiple Disk Systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):142–153, 1995.

[6] S. Melink, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a Distributed Full-Text Index for the Web. *ACM Transactions on Information Systems*, 19(3):217–241, 2001.

[7] R.Freitas and W. Wilcke. Storage Class Memory: The Next Storage System Technology. *IBM Journal of Research and Development*.

[8] B. A. Ribeiro-Neto, E. S. de Moura, M. S. Neubert, and N. Ziviani. Efficient Distributed Algorithms to Build Inverted Files. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Berkeley, CA, USA, August 1999.

[9] C. Stanfill, R. Thau, and D. Waltz. A Parallel Indexed Algorithm for Information Retrieval. In *Proceedings of the 12th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Cambridge, Massachusetts, United States, 1989.

[10] J. Teuhola. A Compression Method for Clustered Bit-Vectors. *Information Processing Letters*, 7(6):308–311, 1978.

[11] A. Tomasic and H. Garcia-Molina. Performance Issues in Distributed Shared-Nothing Information-Retrieval Systems. *Inf. Process. Manage.*, 32(6):647–665, 1996.

[12] A. N. Vo and A. Moffat. Compressed Inverted Files with Reduced Decoding Overheads. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Melbourne, Australia, August 1998.

[13] J. Zobel and A. Moffat. Inverted Files for Text Search Engines. *ACM Computing Surveys*, 38(2), 2006.