Highly Scalable Algorithm For Distributed Real-Time Text Indexing

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

Abstract

Stream computing research is moving from terascale to petascale levels. It aims to rapidly analyze data as it streams in from many sources and make decisions with high speed and accuracy in fields as diverse as security surveillance and financial services including stock trading. We specifically consider real-time text indexing and search with high input data rates (10 GB/s or more) along with small index age-off(expiry) time. This makes it necessary to have maximal indexing rates for large volumes of data as well as minimal latency for indexing (time between start of indexing for a document and its availability for search) while maintaining very-low search response time. In addition, future massively parallel architectures with storage class memories will enable high speed in-memory real-time indexing, where index can be completely stored in a high capacity storage class memory.

In this paper, we present the design of distributed data-structures and distributed real-time text indexing algorithm for parallel systems having large (thousands to hundred thousand) number of cores/processors, while simultaneously providing acceptable search performance [1]. The inherent trade-offs involved in index space, indexing throughput and search response time make this problem particularly challenging. Our algorithm uses group-based index construction and leverages novel index data structures that reduce load imbalance and make text indexing and merge process more scalable and efficient. We show analytically that the asymptotic parallel time complexity of our distributed indexing algorithm, is at least $\Omega(log(P))$ factor better than typical indexing approaches, where P is the number of indexing nodes in a group. We further demonstrate the performance and scalability of our distributed indexing algorithm, on an MPP architecture (Blue Gene/L [1]) using actual IBM intranet data. We achieved high indexing throughput of around 312 GB/min on an 8K node Blue Gene/L machine. In comparison with parallel indexing implemented using typical approaches like CLucene [2], this is 3$\times$ - 7$\times$ better. To the best of our knowledge, this is the first published result on indexing throughput at such a large scale, with sustained search performance. We further show that our approach is scalable to $128K$ nodes, giving an estimated indexing throughput of $5TB/min$. We also achieved indexing latency that is around $10\times$ better than typical indexing approaches.

1. Introduction

Stream computing is a burgeoning area of research and product development driven by customers confronting the exponential growth in the volume of information [3, 4]. It aims to rapidly analyze data as it streams in from many different sources and make decisions with high speed and accuracy. Also, Data-Intensive Super Computing (DISC) is gaining research momentum (see [2]). DISC systems differ from conventional supercomputers in their focus on data: they acquire and maintain continually changing data sets, in addition to performing large-scale computations over the data. With the massive amounts of data arising from such diverse sources as telescope imagery, medical records, online transaction records, and web pages, DISC systems have the potential to achieve major advances in science, health care, business efficiencies, and information access.

In future there will be a strong need for real-time indexing of massive amounts of data flowing at the rate of 10GB/s or more. This data needs to be searched for patterns and the search results are time-critical in fields as diverse as security surveillance, financial services including stock trading, monitoring critical health conditions of patients, climate warning systems and so on. Here, the index will be expected to *age-off* (expire) in a small time and hence will be of bounded size. However, such scenarios cannot tolerate any violation of indexing latency and strict search response times. In addition, future massively parallel (multi-core) architectures with storage class memories [3] will enable high speed in-memory real-time indexing, where index can be completely stored in a high capacity storage class memory. However, current approaches are primarily focused at reducing disk-access time in indexing and its trade-offs with search time ([4], [5], [6], [7], [8]). This motivates exploration of performance optimizations for *in-memory* text indexing and search. Here, we need extremely fast indexing rates and minimal indexing latency along with tight constraints of search response time in large scale systems.

1. http://www.research.ibm.com/bluegene
2. http://www.sourceforge.net/projects/clucene
3. www.almaden.ibm.com/institute/resources/2008/presentations/Halim.ppt
4. http://ati.amd.com/technology/streamcomputing/

The current indexing approaches such as CLucene and single-pass based indexing [9] use inefficient data-structures for merging two index *segments*(segment represents inverted index for a subset of documents in a collection). The data-structures used in such approaches require two expensive operations for merge between two index segments, namely: merge-sort of terms and data re-organization of document and *postings data*(list of positions for the occurrence of a term in a document). Efficient parallel merge sort can help here but expensive data reorganization is unavoidable. Since these data-structures lead to *expensive merge* they are not suitable for real-time indexing with high throughput. Expensive merge leads to *load-imbalance* and hence *poor scalability* in distributed indexing algorithm that uses separate nodes to generate segments and to merge them. Hence, one needs to re-design the index data-structures and ensure they enable efficient merge. For distributed indexing, the algorithm should lead to scalable indexing throughput and low indexing latency while at the sametime sustaining search response time and throughput. This is a challenging problem due to inherent trade-offs between index size, indexing throughput and search response time and throughput. We focus on indexing throughput maximization with same or better search performance and around two times the index size of typical indexing approaches. We present the design of innovative distributed data-structures and algorithm that meet the design goals by enabling efficient index merge that leads to improved load balance and hence better scalability.

It is known that for a single cluster, search is not scalable when performed over more than certain number of nodes in a system depending on workload and system configuration. [1] claims, using experiments and queuing theory based analytical model, that after $2K$ nodes in a single cluster the search performance degrades at a fast rate. After this threshold, we need to have hierarchical indexing and search. In addition, the future multi-core and many-core architectures will have large number of cores where individual core performance may be very small compared to out-of-order cores of current high-performance microprocessors. To provide scalable indexing performance on such multi-core and many-core architectures, we need to have fast hierarchical indexing that scales well with increase in number of cores.

We use the technique of partitioning the total input data across *groups of nodes* to deal with massive amounts of data and simultaneously meet stringent constraints on search response times. Instead of constructing single index per node (*single-node-based* distributed indexing), a single index per group is constructed. This is referred to as ***group-based*** distributed indexing. The *group-based* index is constructed by merging the indexes from each node in the group. The encoding of index data structures helps in improving *cache performance* and also in reducing *communication cost* of the distributed indexing algorithm. The *group-based* index construction algorithm helps in reducing the number of nodes involved in search. This helps in reducing the search response time in large systems, especially in cases where global scoring requires communication between the search nodes. Our *group-based* distributed indexing algorithm is well-suited for massively parallel architectures with large number of cores/processors as it allows scalability to the order of $100K$ nodes. We store the index in memory, eliminating disk access overheads for index construction and search.

This paper makes the following contributions:

- ***Design of distributed data-structures and indexing algorithm***: We designed (architecture independent) novel data-structures and *group-based* in-memory distributed text indexing algorithm which have better load balance and low communication cost and hence improved scalability of distributed indexing. We establish theoretically that the asymptotic parallel time complexity of the distributed algorithm using our data-structures is at least $\Omega(log(P))$ better than the same algorithm using the data-structures of typical indexing approaches, where P is the number of indexing nodes in a group.
- ***Indexing throughput***: We obtained peak indexing rates of around 312 GB/min on $8K$ nodes of BG/L using actual IBM intranet data. To the best of our knowledge, this is the first published result on indexing throughput at such a large scale, with sustained search performance. Based on semi-quantitative analysis, we show that our algorithm is scalable to $128K$ nodes giving an estimated indexing rate of $5TB/min$. We also demonstrate $3\times$ - $7\times$ gain in distributed indexing performance and better strong scalability [5] and weak scalability [6], compared to parallel in-memory implementation of typical indexing approaches like CLucene.
- ***Indexing latency***: We demonstrate $10\times$ better indexing latency than typical indexing approaches on Blue Gene/L. This is due to efficient merge that does not involve data re-organization.

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 and massively parallel multi-core architectures of the future. Moreover, it does not require expensive or fine-grain synchronization.

The rest of the paper is organized as follows. Section 2 gives related work in the area of distributed and real-time text indexing. Section 3 gives the performance drawbacks of typical indexing approaches, presents the design of novel data-structures for indexing, and describes single node index construction algorithm. Next, in Section 4 we

5. Scaling in time with increase in number of nodes while maintaining constant input data size

6. Scaling in time with increase in both number of nodes and data

present detailed design and analysis of the *group-based* distributed indexing algorithm. Section 5 presents the results and analysis of our experiments on Blue Gene/L. Finally, Section 6 concludes with summary and directions for future work.

2. Related Work

In this section we provide an overview of previous efforts in distributed and real-time text indexing. [10] explains a generalized Map-Reduce framework. Here the programs written in functional style are automatically parallelized and executed on a large cluster of nodes. Our scalable in-memory indexing techniques can be implemented as part of the run-time of this framework, which will improve the performance of this system for large-scale indexing. This makes our approach complementary to the general Map-Reduce frameworks.

[11] uses geometric partitioning to construct multiple indexes and provides fast on-line indexing throughput. But it makes a trade-off on search-time although to a limited extent using controlled number of indexes. We provide scalable distributed indexing algorithm without trade-off on search time. [12] also explores the idea of doing less preprocessing of arriving data, at the expense of tolerable latency in the query response time. It targets search systems that rebuild smaller *stop-press* indices once or twice an hour. We target very high indexing throughput of around $5TB/min$ on a large parallel system ($128K$ nodes) with sustained search performance. [13] optimizes real-time index updates in the context of file-system search, using hybrid index maintenance technique that includes logarithmic merge for disk-based indexes and in-place update for long-inverted lists. It uses generalized Zipfian distribution to derive characteristics of the inverted file. We build single in-memory index within an index-group and derive asymptotic time complexity for indexing using document distribution parameters apart from detailed experiments on an MPP system(BG/L). Hadoop [7] uses an optimized file system, HDFS, to provide distributed indexing and search using many nodes where the index resides on disk. We optimize parallel *in-memory* real-time indexing performance where the index resides completely in memory.

[9] provides optimizations for indexing using single-pass approach with limited main memory by creating bit-vectors per term for postings. Their approach uses an expensive merge-sort approach to merge temporary segments on disk to create the final index file. In contrast, we have an efficient segment merge step for in-memory single-pass text indexing.

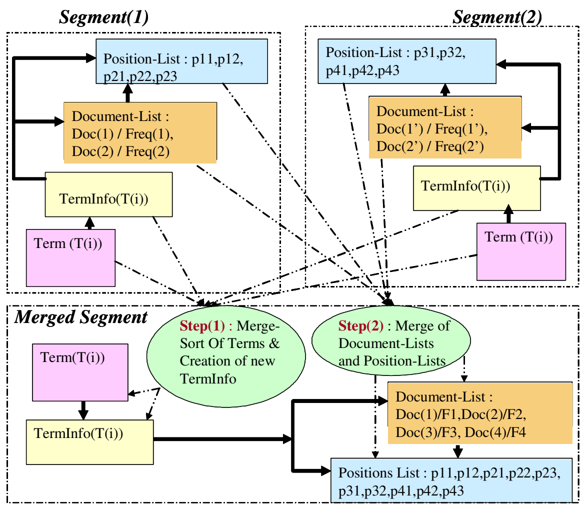
7. hadoop.apache.org/core/

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

In this section, we first study the scalability challenges for indexing, followed by, the design of novel data-structures and algorithm for text indexing and time complexity analysis of our algorithm.

3.1. Scalability Challenges For Indexing

Typical indexing approaches like CLucene and single-pass based indexing [9] involve expensive merge. Fig. 1 shows two input segments, *Segment(1)* and *Segment(2)* that are merged to form the *Merged Segment*. Both the input segments contain a list of *Terms*, each with their *TermInfos*. Each *TermInfo* has an associated *Document-List*(list of documents containing that term and term frequency in the document) and *Position-List*(list of positions per document where that term occurs in the document). The merge involves two expensive operations. First, it involves merge-sort of sorted terms in the input segments to generate the list of Terms and TermInfos for the *MergedSegment*(refer Step(1) in Fig. 1). Second, the *Document-List* and *Position-List* of the input segments are merged and re-organized to form the *Document-List* and *Position-List* of the *Merged Segment*(refer Step(2) in Fig. 1). Hence, parallel indexing algorithms using such approaches have poor scalability as the index-merge process becomes the bottleneck quickly and causes load imbalance. Further, there are inherent trade-offs involved in index size, indexing throughput and search performance which makes the design challenging.



Segment(1)
Segment(2)
Position-List : p11,p12, p21,p22,p23
Document-List : Doc(1) / Freq(1), Doc(2) / Freq(2)
TermInfo(T(i))
Term (T(i))
Position-List : p31,p32, p41,p42,p43
Document-List : Doc(1') / Freq(1'), Doc(2') / Freq(2')
TermInfo(T(i))
Term (T(i))
Merged Segment
Step(1) : Merge-Sort Of Terms & Creation of new TermInfo
Step(2) : Merge of Document-Lists and Position-Lists
Term(T(i))
TermInfo(T(i))
Document-List : Doc(1)/F1,Doc(2)/F2, Doc(3)/F3, Doc(4)/F4
Positions List : p11,p12,p21,p22,p23, p31,p32,p41,p42,p43

Figure 1. Lucene index merge process

3.2. Two-level Hierarchical Index Data Structure Design

The design for the index data structure considers efficient index merge while sustaining the search performance including search throughput and search response time. The key idea for eliminating index merge overheads is to use a two-level hierarchical data structure based index representation. 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 set of second-level hash tables. The second-level hash-table, called ***IHT*** (Document Interval Hash Table) is an index for a set(interval) of documents with contiguous Document IDs(documents numbered from 1 to D). Each term in IHT is mapped to a list of document IDs, where each document contains that term. For each such document the detailed occurrence positions called postings data is stored.

We note that our two-level hierarchical data-structure is different from a standard two-level hash-table where for the first level hash table the key is a term and the value is another hash-table. This hierarchical data structure based approach gives two key advantages over single level hash-table approach and sorted list of terms based approach as used in CLucene. First it avoids the need for repeated re-organization of data in the IHTs that get merged in the final merged index (GHT) as it only involves adding a reference to the IHT in GHT for terms contained in that IHT (details described in the sections below). Second, the terms in both IHT and GHT are organized in a hash-table structure and hence don't require merge-sort during merging.

3.2.1. IHT Design. The IHT represents an index for documents with contiguous IDs (documentID $\in [1..D]$). Fig. 2 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. In the implementation, the IHT is *serialized* and stored in an *encoded* format(for details refer [14]). The serialization enables *good cache* performance, lesser memory consumption by enabling compression techniques and low *communication cost* for distributed indexing. The encoding preserves hash-table based access for document and positions data for a given term. This helps in keeping search efficient while at the same time enabling efficient merge of IHT into GHT.

3.2.2. GHT Design. The GHT (Fig. 3) contains a hash-table where the key is a unique term in the document collection and the value is a list of IHT numbers, where each IHT has at least one document that contains that term. This design leads to low memory for GHT and fast search using it. Our index also has an array called the Array-of-IHTs whose each entry

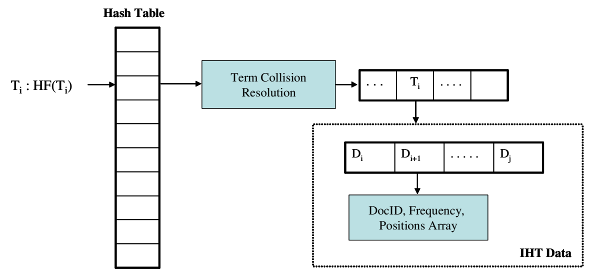


Figure 2. Interval Hash Table (IHT)

points to the encoded IHT corresponding to that document interval. Since, the document IDs in an IHT are local to it, we also keep a base document ID with each entry in the Array-of-IHTs. The global document ID is computed by adding the base ID to the document ID in the IHT.

3.3. Indexing Algorithm

Our indexing algorithm has three main steps:

- Posting table (LHT) for each document is constructed without involving sorting of terms;
- Posting tables of k documents are merged into an IHT, which are then encoded appropriately;
- Encoded IHTs are merged into a single GHT in an efficient manner.

For IHT construction, first, the posting table (also referred to as ***LHT***) for each of the documents is formed separately without involving sorting of terms. Then, each set of k posting tables(LHTs) are merged into one IHT per set. From each LHT, each unique term is read one-by-one and inserted into the IHT, if it is a new term. Then, the document and postings data for this term is merged into the already available document and postings data for the term in the IHT. IHT helps in scalable distributed indexing by providing the ability to offload the construction of index for a set of documents with contiguous IDs to another processor before merging that into GHT.

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. 3, the entry g points to $IHT_{(g)}$. Also, store a base document ID for this IHT. The base ID is the sum of the base ID of the previous IHT and the number of documents in it (or the document interval size, if all IHTs contain same number of documents).
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. 3 illustrates how $IHT_{(g)}$, is merged into the GHT. First (Step-*S1* in Fig. 3), $IHT_{(g)}$ is pointed to by the appropriate location in the Array-of-IHTs. Then (Steps-*S2(a) & S2(b)* in Fig. 3), $IHT_{(g)}$, 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, in contrast, to typical indexing approaches which re-organize the segment data when merging it into the final merged segment. This makes GHT/IHT design efficient for distributed indexing.

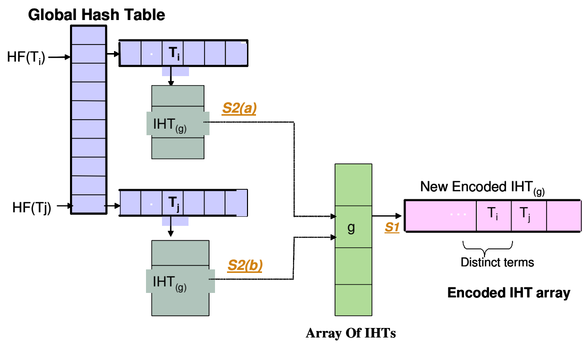


Figure 3. GHT structure and construction from IHT

3.4. Search using GHT/IHT data structures

The sequential search algorithm for a query term using our GHT/IHT data structures has following three steps: (a) GHT is searched for the query term, using typical hash-table operations, to obtain the list of IHT numbers and the corresponding IHT data. (b) Each IHT is searched, using hash-table based access, for the above term to get the list of document IDs (and the corresponding term-frequencies) containing that term (refer Fig 2). (c) The results of each IHT are combined to get the list of document IDs containing that term in the full index. These steps demonstrate that the search algorithm used to retrieve the document IDs for a term using IHT/GHT based hierarchical data structure is efficient. In case of distributed search, our index data-structures enable parallel matching document search for a term on IHTs at different nodes with low communication overheads. This leads to potentially better distributed search performance over single-node based parallel indexing and search(refer section 5.4). Further, as mentioned in section 1, this *group-based* indexing approach using IHT/GHT index data-structure results in sustained search performance over hundred thousand nodes which is not possible in a *single-node* based approach.

3.5. Asymptotic Time Complexity of S_{ghtl}

This section presents the time complexity of the core phases in the text indexing algorithm using our novel data-structures. The sequential version of the text indexing algorithm is referred to as S_{ghtl} in the paper.

We don't use typical IR models like Generalized Zipfian distribution [13] for term-distribution in text collection, or, Heaps's Law [13] for size of an active text vocabulary. Instead, we present the time complexity in terms of parameters at a finer level of granularity as they can vary independently depending on the nature of the document collection. These are as follows:
"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).
"$\gamma(k)$": 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.
"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. Both γ and θ vary with k, number of documents considered in a set. Since, α and β are per-document parameters, averaged appropriately, they are treated as constants in the paper.

3.5.1. IHT Construction: Time Complexity Analysis.

Since LHT construction involves typical hash-table operations its complexity is proportional to α. When IHT is formed from k LHTs, the final unique list of terms ($O(\gamma)$) is kept efficiently in a hash-table. The work involving copying term-postings per document and term is proportional to the postings data being processed. Hence, the work in this merge phase is proportional to $[(\alpha * k * \beta) + \gamma]$. The following equation represents the IHT production time:

$$T(IHT\,production) = O(\alpha * k * \beta + \gamma) \tag{3.1}$$

3.5.2. GHT Construction: Time Complexity Analysis.

During GHT construction from IHT, we do not need to perform work proportional to number of documents in an interval (i.e. k), as in typical indexing approaches. Here, we simply take the constructed IHT and insert a pointer to it in the Array-of-IHTs. The work mentioned in Fig. 3 is proportional to number of unique-terms in an IHT i.e. γ. The hash table insert operations are proportional to δ. Hence, the GHT construction time is given by the following equation:

$$T(IHT\,merge) = O(\gamma * R/k + \delta) \tag{3.2}$$

Assuming α and β as constants, adding (3.1) and (3.2) and simplifying, we get the time complexity of index construction on single node, S_{ghtl} as:

$$T(S_{ghtl}) = O(R/k * \gamma) \tag{3.3}$$

Typical indexing approaches (referred to as S_{orgl}) do merge-sort and index data re-organization during the index-merge phases, therefore their time complexity is higher. Below is the equation that represents this time complexity (for details refer [14]):

$$T(S_{orgl}) = O(R*\alpha*(\log(\alpha*k)+\beta)+R/k*\gamma*(\log(R/k)+\theta)) \tag{3.4}$$

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

$$T(S_{orgl}) = O(R/k * \gamma * (\log(R/k) + \theta)) \tag{3.5}$$

4. Distributed Algorithm For Text Indexing

In this section we explain the *group-based* distributed indexing algorithm. The distributed algorithm using typical indexing approaches is referred to as P_{orgl} while that using our data-structures and merge approach is referred to as P_{ghtl}.

4.1. 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 single index for each node in the system(referred as *single-node-based* approach), as is common in most distributed indexing approaches, we actually create one index per index-group which results in distributed index for the complete set of documents in the system (hence the term *group-based* distributed indexing algorithm).

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 tree-based merge procedure.

In case of P_{ghtl}, the Producers, construct encoded IHTs. All Producers store the encoded IHTs locally and for each IHT, a Producer generates the list of terms and the document frequency per term in that IHT(referred to as the ***IHT-meta-data*** for that IHT). Since the Producers don't send the complete IHT and instead send only the IHT-meta-data, it helps in saving space at the Consumer node and also *reduces communication cost*. 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. 4. Efficient merge at the Consumer node leads to *better load balance* for P_{ghtl} as compared to P_{orgl}.

For search, in P_{ghtl}, same query is provided to each Consumer (index-group head) that has the merged index. Each Consumer determines the matching documents for that query by communicating with the Producers that have the IHTs which contain the document information for the terms in the query. The Consumers share matching document information for global scoring. This is followed by scoring and selection of top-N documents by each Consumer; and finally, across-Consumer top-N document selection and reporting to the user. The search in case of P_{orgl} is similar except that the each Consumer has full index for its group to perform search. P_{ghtl} enables parallel search for matching documents for a term with low communication overhead it can lead to lower distributed search time compared to P_{orgl}.

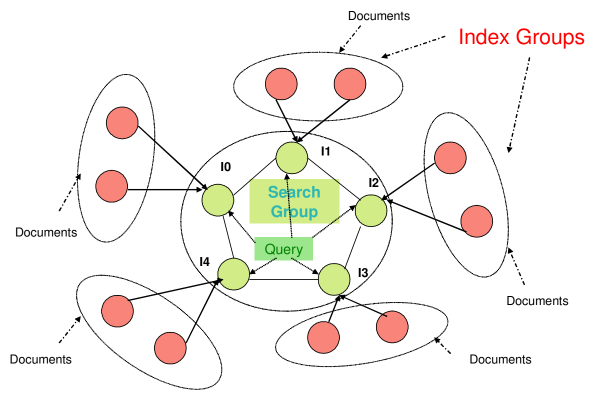


Figure 4. Distributed Indexing & Search

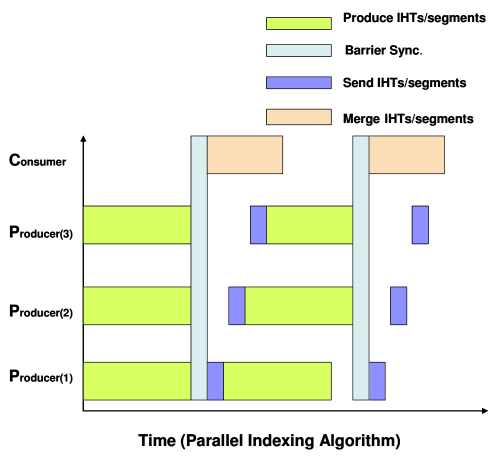


Figure 5. Distributed Indexing Pipeline Diagram

The pipeline diagram in Fig. 5 illustrates the steps involved in distributed indexing. These overall steps are applicable in both cases - P_{ghtl} and P_{orgl}. These are:

- (a) Generation of encoded IHTs(P_{ghtl})/segments (P_{orgl}), in parallel, at the Producers;
- (b) Communication of IHT-meta-data/segments from the Producers to the Consumer;
- (c) 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.

4.2. 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. 5 to analyze the time complexity of both P_{ghtl} and P_{orgl}. Let there be n rounds with P Producers in each round (also referred to as ***produce-consume*** 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. Similarly, $Cons_{(i)}$ denotes the total time spent by the Consumer in the i^{th} round which includes both the compute time for merging and its communication time in i^{th} round. The distributed indexing time is approximately given by the following equation :

$$T(distributed) = X + Y + Z \tag{4.1}$$

$$where, X = \max_{j} ProdComp_{(j,1)} \tag{4.2a}$$

$$Y = \sum_{2 \leq i \leq n} \max(\max_{j} Prod_{(j,i)} \, , \, Cons_{(i-1)}) \tag{4.2b}$$

$$Z = Cons_{(n)} \tag{4.2c}$$

We use equations (3.1) and (3.2) for IHT production and merge time in the above equation for P_{ghtl} time, and ignore the communication time as it is small in our experiments (with index-group size $<= 128$). 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 these two cases (for details refer [14]):

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

$$T(P_{ghtl}) = O(R * \alpha * \beta/P) = O(R/P) \tag{4.3}$$

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

$$T(P_{ghtl}) = O(R * \gamma/k) \tag{4.4}$$

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. Additionally using term-based partitioning gives even better scalability but we omit these details for brevity.

P_{orgl} has an expensive merge phase and worse asymptotic time complexity and is less scalable compared to P_{ghtl}. We developed the time complexity for P_{orgl} and observed time complexity for Case(1) as $O((R/P) * log(k))$ and for Case(2) as $O((R * \gamma/k) * log(P))$. This proves that the parallel time complexity of P_{ghtl} is at least $\Omega(log(P))$ better than P_{orgl} (for details refer [14]).

5. Results and Analysis

5.1. Experimental Setup

We implemented our GHT/IHT based indexing data-structures and algorithm, P_{ghtl}, on the original CLucene codebase (v0.9.20). We also implemented CLucene based distributed indexing algorithm, P_{orgl}, where we maintain the index in memory using the *RAMDirectory*. We used CLucene for comparison due to its easy availability and its wide acceptance in the open source community.

We studied the scalability of P_{orgl} and P_{ghtl} by doing experiments on IBM intranet website data as used in [1]. The text data was extracted from HTML files and loaded equally into the memory of the producer nodes, *before*, the indexing time measurement is started. For P_{orgl}, we used a value of k so that only one segment is created from all the text data fed to a Producer so as to get its best indexing throughput. For indexing latency measurement we had to however choose the same lower value of $k = 256$ as used for P_{ghtl}, so that we could compare it meaningfully with P_{ghtl}.

The experiments were conducted on the BlueGene/L platform as it was readily available with large configurations(up to 16K nodes). BG/L 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 also evaluated the L1 Icache and L1 Dcache performance of our indexing algorithm on a single node using the Dinero cache simulator [8]. We ran the experiment for indexing 100MB of data and got 99% hit-rates for both L1-Icache and L1-Dcache, as in CLucene.

5.2. Indexing Throughput Analysis

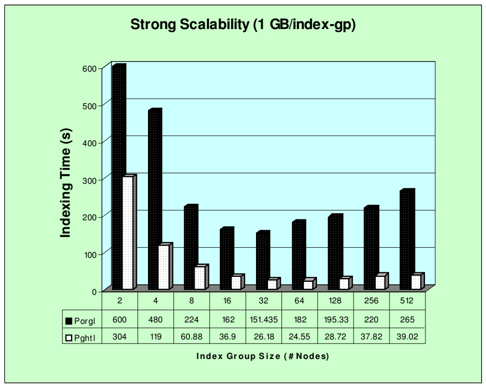
In this section we study the scalability of indexing throughput with variation in number of processor nodes and input text data size.

5.2.1. Strong Scalability Study. In the strong scalability experiment, the input data size for an index group remains constant while the size of the group is increased. We

8. http://pages.cs.wisc.edu/ markhill/DineroIV/

consider index group size of G processors with P Producers and 1 Consumer, with group size varying from 2 to 512 nodes. We indexed large volumes of data up to 256 GB, by having multiple index-groups and 1 GB text data per group.

The plots of strong scalability study for P_{orgl} and P_{ghtl} are given in Fig. 6. 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. 7 shows the variation of speedup as the number of processors is increased. The maximum speedup (relative to G = 2) 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 6. Strong Scalability

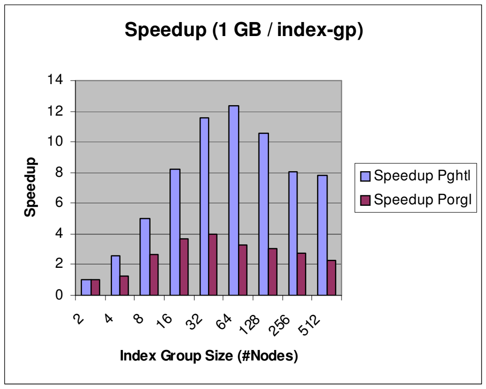


Figure 7. Speedup (strong scalability)

This behavior can be explained by the inefficient merge process of P_{orgl} that leads to load-imbalance and hence poor performance and scalability. 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 4.2 and equation (4.3) for P_{ghtl}. However, as the index-group size increases, the 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 4.2 and equation (4.4) for P_{ghtl}. Since, merge is performed by a single consumer node it becomes a bottleneck for scalability due to load imbalance. 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. We have observed even better scalability of P_{ghtl}, upto $G = 512$, using term-based partitioning but we leave these details for brevity. We get peak indexing rate within a single-index group(G = 64) of about 2.44 GB/min. Now assuming search is scalable to around 2K nodes [1], if we have 2K such independent index-groups, each of size 64 nodes, we will get a peak indexing rate of around 5 TB/min, while maintaining acceptable search performance. As part of our experiment, we instead used 8K nodes and got a peak indexing rate 312 GB/min. We can obtain even better indexing throughput by implementing sequential code optimizations but leave this for future work.

5.2.2. 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. 8, as G increases from 4 to 128, P_{ghtl} time increases from 6.64s to 37.92s, i.e. by 5.71$\times$ factor, while P_{orgl} time increases from 15s to 290s, i.e. by 19.4$\times$ 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}.

5.2.3. Scalability with increase in data size. In this experiment we study the indexing time variation with increase in text data size for a constant group size. Here the size of text data used to generate index in one index group is varied from 64MB to 1GB. Again, P_{ghtl} performs better than P_{orgl} (Refer Fig 9).

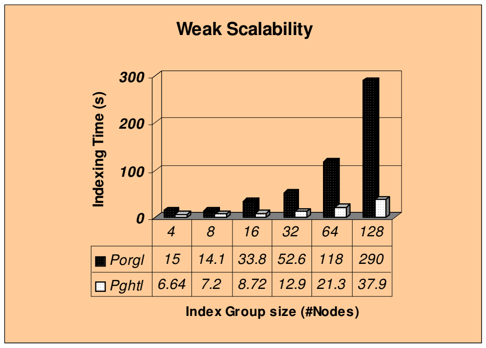


Figure 8. Weak Scalability Study

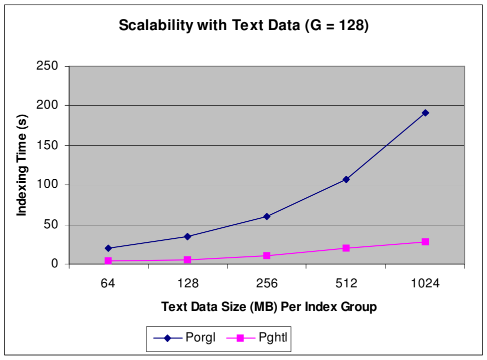


Figure 9. Scalability with increasing text-data (G = 128)

5.3. Indexing Latency Analysis

We measure the indexing latency as the time from the start of document indexing to the time the index for the document is available for search.

Fig. 10 displays the variation in indexing latency with increase in size of the group, G. The amount of data indexed per group is maintained constant at 1 GB and the number of documents per IHT in P_{ghtl} is kept as 256, and is the same as number of documents per first-level segment in P_{orgl}. We can see that the indexing latency for P_{ghtl} is one order of magnitude(around 9.91x) better than P_{orgl}. The indexing latency is dependent on the segment merge time at the Consumer node in case of P_{orgl} and since this is inefficient it leads to a larger latency and this increases with the increase in the number of segments to merge. In case of P_{ghtl} the merge of IHT-meta-data is much more efficient and hence it is better.

5.4. Distributed Search Performance

We measured the distributed *search* performance on BG/L using two experiments:

1) Using single index-group of size increasing up to 128 nodes that indexed 2GB of text data;

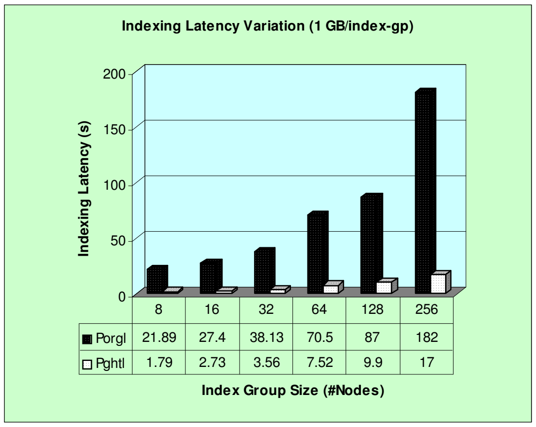


Figure 10. Indexing Latency Variation

2) Using multiple index groups of same size(8) with total number of nodes increasing up to 512 that indexed total 8GB of data.

Both experiments used 1000 queries from query-set used in [1]. For the first experiment P_{ghtl} got better or similar search performance (2.34s for 64 nodes/group, 1.75s for 32 nodes/group) compared to P_{orgl} (2.85s for 64 nodes/group, 2.13s for 32 nodes/group). In the second experiment, for P_{ghtl}, we observed sustained search performance (0.94s for 64 nodes, 1.55s for 512 nodes). P_{ghtl} enables parallel matching document search for a term, in an index-group, on different nodes that have IHTs with low communication overhead. P_{orgl} however performs sequential search for matching docs for a term in a single-group. Hence, for smaller group size P_{ghtl} has lower search time as compared to P_{orgl}.

6. Conclusions and Future Work

Data Intensive Supercomputing at terascale and petascale levels has opened up challenging research problems in distributed algorithm design. Towards this end, we have delivered high throughput text indexing which has been demonstrated for the first time at such a large scale. We presented the design (architecture independent) of new data-structures and algorithm for distributed in-memory real-time *group-based* text indexing which has better load balance, low communication cost and good cache performance. We proved analytically that the parallel time complexity of our indexing algorithm is at least $\Omega(log(P))$ better asymptotically compared to typical indexing approaches. We have demonstrated around $3\times$ - $7\times$ improvement in indexing throughput and around $10\times$ better indexing latency compared to typical indexing approaches on Blue Gene/L. We achieved peak indexing throughput of 312 GB/min on 8K Blue Gene/L nodes and extrapolate this to show that for $128K$ nodes we could achieve 5 TB/min, with acceptable search performance. As part of the future work, we plan to study sequential indexing optimizations and distributed search algorithm optimizations.

References

[1] J. E. Moreira, M. M. Michael, D. D. Silva, D. Shiloach, P. Dube, and L. Zhang, "Scalability of the Nutch Search Engine," in *Proceedings of the 21st Annual International Conference on Supercomputing*, Seattle, Washington, 2007.

[2] R. E. Bryant, "Data Intensive Supercomputing: The Case for DISC," Computer Science Department, Carnegie Melon University, Tech. Rep. CMU-CS-07-128, May 2007.

[3] R. F. Freitas and W. W. Wilcke, "Storage Class Memory: The Next Storage System Technology," *IBM Journal of Research and Development*, vol. 52, no. 4/5, 2005.

[4] J. Zobel and A. Moffat, "Inverted Files for Text Search Engines," *ACM Computing Surveys*, vol. 38, no. 2, 2006.

[5] L. A. Barroso, J. Dean, and U. Hölzle, "Web Search for a Planet: The Google Cluster Architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, 2003.

[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*, vol. 19, no. 3, pp. 217–241, 2001.

[7] K. J. Rao, "Cache Conscious Indexing For Decision Support In Main-Memory," Columbia University, Tech. Rep., 1998.

[8] S. Büttcher and C. L. Clarke, "Indexing Time vs. Query Time Trade-offs in Dynamic Information Retrieval Systems," University of Waterloo, Ontario, Canada, Tech. Rep. CS-2005-31, Oct 2005.

[9] S.Heinz and J. Zobel, "Efficient Single-Pass Index Construction For Text Databases," *Jour. of the American Society for Information Science and Technology*, vol. 54, no. 8, pp. 713–729, 2003.

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

[11] A. M. N. Lester and J. Zobel, "Fast OnLine Index Construction by Geometric Partitioning," in *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, Bremen, Germany, 2005.

[12] R. Lempel, Y. Mass, S. Ofek-Koifman, D. Sheinwald, Y. Petruschka, and R. Sivan, "Just In Time Indexing for up to the Second Search," in *Proceedings of the sixteenth ACM conference on Conference on Information and Knowledge Management*, Lisbon, Portugal, 2007.

[13] S. Büttcher, "Multi-User File System Search," Ph.D. dissertation, University Of Waterloo, Canada, 2007.

[14] A. Narang, V. Agarwal, V. Garg, and M. Kedia, "Scalable Algorithm Design for Parallel Hierarchical Text Indexing," IBM Research, Tech. Rep. RI08009, June 2008.