Distributed Architecture for Information Retrieval

Claudine Badue[1] Ricardo Baeza-Yates[2] Wagner Meira[3] Berthier Ribeiro-Neto[4]
Nivio Ziviani[5]

{claudine,meira,berthier,nivio}@dcc.ufmg.br rbaeza@dcc.uchile.cl

[1,3,4,5]Federal University of Minas Gerais
Belo Horizonte, Brazil

[2]University of Chile
Santiago, Chile

ABSTRACT

We propose a random layout for distributed information retrieval systems based on a global index. The random layout is used to implement a random global index, which allows balancing the load and attaining improved performance. The distributed system adopts a network of workstations model and the client-server paradigm. Documents are ranked using the vector space model along with a document filtering technique. In the random global index, a global inverted file is generated for all documents in the text database and fixed size blocks of the inverted lists are randomly distributed among processors. Using a real Web data collection, we compare its performance against two other index partitioning schemes, namely lexicographical global index and local index. For this, an analytical model coupled with a simulator is developed and validated. Regarding load balance, the random layout outperforms the lexicographical with gains reaching two times. Regarding response time, the random layout and the lexicographical have similar performance on average. Both layout schemes outperform significatively the local index, with gains that reached five times. Due to its better load balancing, competitive query processing performance and flexibility for system reconfiguration, we believe that global indexes based on random layouts are a good choice for the design of large distributed information retrieval systems.

1. INTRODUCTION

Electronic commerce is a rapidly growing area on the World Wide Web. Applications on electronic business must be able of serving an increasing number of suppliers and customers with an expanding commercial data content. Furthermore, web-based commerce demands for very high levels of efficiency, availability and scalability. To achieve such efficient and dependable service, underlying information retrieval systems can turn to distributed and parallel storage and searching.

To find documents quickly, information retrieval systems build inverted indexes on disk. Two approaches have been proposed in the work presented in [17] to distribute the inverted index among various computers. They are the local index (LI) and lexicographical global index (LGI). In the local index, the documents in the text database are distributed among the processors, and each processor generates an inverted file for its documents. In the lexicographical global index, an inverted file is generated for all the documents in the text database and the inverted lists are distributed among processors according to a lexicographical order of terms.

Generating and maintaining local indexes is simple because all the work can be done locally without interaction among the processors. However, each processor has to execute the whole query, which provides high parallelism but also degrades performance considerably. In the lexicographical global indexes, the terms of a query are sent only to the processors that hold the related inverted lists, which provides high concurrency. Nevertheless, this scheduling scheme might deteriorate load balance, because the processor holding the most frequent terms in a query is heavily loaded.

The objective of this paper is to present a new layout for organizing a global index for distributed query processing, called random global index (RGI). We divide the inverted lists in blocks of the same size and randomly select a processor to hold each block of each inverted list. This random layout for the global index combines the advantages of the local and lexicographical global index partitioning schemes because it provides a better load balance among processors and allows a good degree of query concurrency within the system. As inverted lists are randomly distributed among processors, the execution of a query is not restricted to a processor, which provides for parallelism. Not all processors hold blocks of the same inverted list, which allows concurrency. Further, random layouts are simple to maintain; if processors are added to the system, it is much easier to redistribute the data. As we shall see, these effects favor the random global index organization in detriment of the local index and lexicographical global index organizations.

The distributed system uses a network of workstations model. The workstations are tightly coupled by fast network switching technology. The retrieval system adopts the client-server paradigm that consists of a set of processors and a designated broker. An analytical model coupled with a simulator was developed and validated, in order to evaluate query performance in our distributed text database for more system configurations than are currently available.

Our results show that the random global index is competitive and sometimes outperforms the traditional lexicographical global index and local index techniques, considering response time, throughput, load balance, and processing cost. We used a Web data collection for performance measurement. The random global index and lexicographical global index have similar performance on average in response time, and both outperformed significatively the local index, where the gains reached five times approximately. A similar result is observed regarding the system throughput.

This paper is organized as follows. Section 2 covers the related work. Section 3 presents the distributed text database, describing the system architecture, the index structure, the vector space model as ranking strategy and the query processing. Section 4 describes an analytical model for predicting distributed query performance. Section 5 shows the experimental results, and Section 6 presents the conclusions and future work.

2. RELATED WORK

The work presented in [17] proposes the two basic and distinct options for storing the inverted lists, namely local index and lexicographical global index. The work in [7] considers the two basic schemes for index partitioning proposed in [17] for a shared-everything multiprocessor machine with multiple disks. The work in [11] considers the two index organizations proposed in [17] for a tightly coupled network, and investigates how query performance is affected by the index organization, the network speed, and the disks transfer rate. Our work differs from those presented in [17, 7, 11] in the following aspects. First, while the works in [17, 7] adopt the boolean model, we use the vector space model. Second, while the works in [17, 7] model documents and queries, and the work in [11] uses documents and queries in the TREC-3 collection [5], we base our experimental results on a Web data collection maintained by the TodoBR [16] search engine. Third, while the works in [17, 7] consider only a sequential query service, we address a concurrent query service. Fourth, none of these works have used a random index allocation.

In the works presented in [9, 1], the two traditional types of index partitions proposed in [17] are investigated using a real distributed architecture implementation. Our work differs from those presented in [9, 1] in the following aspects. First, while the work in [9] uses part of the documents and queries in the TREC-7 collection [6], and the work in [1] employs documents and queries in the TREC-3 collection [5], we base our experimental results on a real Web data collection. Second, while they implement a real case framework, we develop and validate an analytical model in order of evaluating performance for more system configurations than are currently available. Third, while the work in [9] addresses only a sequential query service, we consider a concurrent query service. Fourth, none of these works have used a random index allocation.

The works presented in [14, 8] compare performance of a multimedia storage server based on a random data allocation layout with traditional data striping techniques. Random data allocation for multimedia servers has also been considered in [15, 3, 4]. The work in [15] analyzes the performance of a clustered video server with random allocation of data blocks. In the works in [3, 4], random data allocation is considered on RAID systems for 3D interactive applications. Our work differs from those presented in [14, 8, 15, 3, 4] because while we evaluate performance of a textual query processing system, they investigate performance for multimedia systems, that have different requirements and workloads from ours.

3. DISTRIBUTED TEXT DATABASE

The distributed system uses a network of workstations model, as illustrated in Figure 1. The workstations are tightly coupled by fast network switching technology. Each workstation has its own local memory and local disk. The advantages of this shared nothing model are that all communication between processors is done through messages, which eliminates interference from operating system memory control processes, and that disks are directly accessed by processors without going through the network.

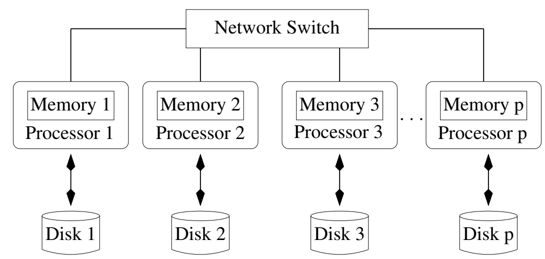

Figure 1: Network of workstations model.

The retrieval system adopts the client-server paradigm that consists of a set of processors and a designated broker, responsible for accepting client queries, distributing the queries to the processors, collecting intermediate results from the processors, combining the intermediate results into the final result and sending the final result to the client. Each of the processors and the broker runs on a separate machine. Figure 2 illustrates the client-server paradigm.

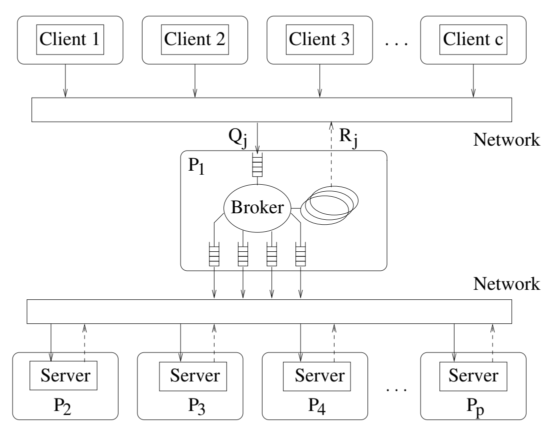

Figure 2: Client-server paradigm.

The text database is indexed using the inverted file technique [2, 18]. An inverted file is an indexing structure composed of two elements: the *vocabulary* and a set of *inverted lists*. The vocabulary contains each term t in the text document collection; the terms are sorted in lexicographical order. There is one inverted list for each term t, consisting of the identifiers of the documents containing the term and, with each identifier d, the frequency $f_{d,t}$ of t in d. Thus, inverted lists consist of term entries, that is, pairs of $< d, f_{d,t} >$ values. As we adopt the vector space model along with a technique for filtering documents during ranking, the inverted lists are sorted by decreasing within-document frequency [10].

3.1 Index Partitioning

As mentioned, in this paper we compare three partitioning schemes for the index, as a strategy for answering queries faster:

Local index (LI): each processor generates an inverted file for its local documents.

Lexicographical global index (LGI): a global inverted file for the whole text database is generated, and the inverted lists are distributed among processors in lexicographical order. According to this strategy, one possible partitioning for the global index might be one in which processor 1 holds the inverted lists for all the terms that start with the letters A, B and C; processor 2 holds the inverted lists for all the terms that start with the letters D, E, F and G; and so on, such that each processor holds a portion of the global index with approximated size. More details on the performance of the local and lexicographical global indexes may be found in the work presented in [1, 11].

Random global index (RGI): an inverted file is generated for all documents in the text database and fixed size blocks of the inverted lists are randomly distributed among processors.

3.2 Ranking with the Vector Space Model

The documents in the text database collection are ranked using the vector space model [12]. In the vector space model, documents and user queries are represented as vectors of the weight of terms. We assign the weight to a term in a document or a query using the *tf-idf* scheme [13]. The standard algorithm for ranking documents uses a set of accumulators, one accumulator for each document in a collection, and a set of inverted lists. For each query term t, the contribution made by the term t to the degree of similarity between the query q and each document d in the inverted list is added to the document d's accumulator's value. The final result is composed by the documents with the highest accumulator values.

The work in [10] proposes a technique for filtering documents during ranking which allows a significant reduction of ranking evaluation costs without degradation in retrieval effectiveness. In the sequential algorithm the filtering mechanism uses thresholds that are determined as a function of the accumulated partial similarity of the currently most relevant document S_{max}.

In the lexicographical and random global indexes, when processors receive only a few terms, the value of S_{max} is a fraction of that in the sequential algorithm. In the local index, if one of the processors holds only a few high weighted documents, the rising of S_{max} is low. Consequently, the amount of pruned resources in the distributed algorithms is smaller than in the sequential algorithm, which might deteriorate the performance of the distributed algorithms.

The work in [11] proposes a solution to this problem that previews the rising of the S_{max} value before query processing. In this way, the pruning thresholds used during ranking evaluation can be previously calculated by the broker and distributed to the processors, along with the query. This strategy equalizes the pruning of term entries processed from the distributed index, no matter how it is partitioned. By adopting this adaptation of the filtering technique to the distributed processing, we obtain approximately the same effectiveness as the standard algorithm of the vector space model, for all the index partitioning strategies, upon significant reductions in ranking evaluation cost.

3.3 Distributed Query Processing

In this section we discuss the characteristics of a distributed information retrieval system, which consists of a set of processors and a designated broker, each running on a separate machine. The broker is responsible for scheduling the queries to the processors, receiving the intermediate results returned by each one of the processors and combining the intermediate results into the final result. Next, we describe the query processing algorithms implemented in the broker, which differ according to the index partitioning strategy.

Local index (LI): In the local index, the broker sends the query to all processors. Each processor retrieves the documents related to that query in the local subcollection and ranks them; selects a number of documents from the top of the ranking; and returns them to the broker as the local answer set. The broker uses a multiway merge [18] to fuse the local answer sets and produce the final ranked answer set.

Lexicographical global index (LGI): In the lexicographical global index, the broker determines which processors hold inverted lists relative to the query terms, breaks the query into subqueries and sends them to the respective processors. Once a processor has received a subquery, it retrieves the documents related to its subquery and ranks them; selects a number of documents from the top of the ranking; and returns them to the broker as the local answer set. The broker adds the weights of the documents which are present in more than one local answer set and perform a sort using the quicksort algorithm to produce the final ranked answer set.

Random global index (RGI): In the random global index, the broker determines which processors hold blocks of the inverted lists relative to the query terms, breaks the query into subqueries and sends them to the respective processors. For selecting the processors which might be involved in the execution of a query, the broker pre-calculates the pruning thresholds to be used during ranking evaluation of that query, and examines which processors hold blocks with frequencies higher than the thresholds; a processor holding a block with low frequencies is not scheduled for that query. Once a processor has received a subquery, it retrieves the documents related to its subquery in the local subset of blocks of inverted lists and ranks them; selects a number of documents from the top of the ranking; and returns them to the broker as the local answer set. The broker adds the weights of the documents which are present in more than one local answer set and do a sort using the quicksort algorithm to produce the final ranked answer set.

For both global indexes (lexicographical and random), local rankings are less precise, which complicates the cutting strategy that consists of the selection of a number of documents to be sent back to the broker. To solve this problem, we adopted the cutoff factor presented to the lexicographical global index in [11], given by $p \times f2$, where p is the number of processors, $f2 = 5 \times f_1$ and f_1 is the number of documents in the final answer set. Using such factor, they observed no significant variation in the final answer precision.

4. ANALYTICAL MODEL

To evaluate query performance in our distributed text database for more system configurations than are currently available, an analytical model coupled with a simulator was developed and validated. Despite its simplicity, the model perfectly captures the key workload variables and system parameters that determine processing time in our system, with a considerably low estimated deviation from real time.

4.1 Variables and Parameters

Next, we define the notation for the basic workload variables and critical system parameters that cause impact on our system performance.

Workload Variables:

a_i: sum of the number $a_{i,j}$ of pairs <document, similarity> in the local answer sets returned by processors j for query $\vec{q_i}$, that is, $a_i = \sum_{j=1}^{p} a_{i,j}$
$a_{i,j}$: number of pairs <document, similarity> in the local answer set returned by processor j for query $\vec{q_i}$
c_{i,a_i}: number of operations (comparison and swap) during merging of local answer sets for query $\vec{q_i}$ at the broker
$c_{i,j}$: number of operations (comparison or swap) during ranking of documents for query $\vec{q_i}$ at processor j (in local index)
$cs_{i,j}$: number of operations (comparison or swap) during ranking of documents for subquery $\vec{q}_{i,j}$ at processor j (in lexicographical and random indexes)
ct_i: time between the arrival of query $\vec{q_i}$ in the system and the beginning of its processing in any of the processors
$ct_{i,b}$: time between the arrival of local answer sets in the broker and the beginning of their merging
$h_{i,j}$: number of pairs <document, similarity> inserted or accumulated in the set of accumulators during ranking of query $\vec{q_i}$ at processor j
$g\ell_k$: size (in number of blocks of 4 kilobytes) of the global inverted list for term k (in lexicographical index)
$g\ell_{k,j}$: size (in number of blocks of 4 kilobytes) of the global inverted list for term k at processor j (random index)
$\ell\ell_{k,j}$: size (in number of blocks of 4 kilobytes) of the local inverted list for term k at processor j (in local index)
p: number of processors
q_i: number of terms in query $\vec{q_i}$
$\vec{q_i}$: vector of terms of query i
$q_{i,j}$: number of terms in subquery $\vec{q}_{i,j}$
$\vec{q}_{i,j}$: vector of terms extracted from query i and sent (by the broker) to the processor j
t_i: total time (in seconds) to process query $\vec{q_i}$
$t_{i,j}$: time (in seconds) to process subquery $\vec{q}_{i,j}$ at processor j
tf_i: finishing time of processing of query $\vec{q_i}$ in any of the processors
ts_i: beginning time of processing of query $\vec{q_i}$ in any of the processors

System parameters:

b: number of bytes of a pair <document, similarity>
f_1: number of pairs <document, similarity> from the top of the ranking which are returned as local answer set (in local index)
f_2: proportionality constant which affects the number of pairs <document, similarity> from the top of the ranking which are returned as local answer set (in lexicographical and random indexes)
t_c: average time (in seconds) to execute an operation of comparison or swap during ranking of documents
t_h: average time (in seconds) to insert or accumulate a pair <document, similarity> in a hash table
t_r: average time (in seconds) to read a block of 4 kilobytes from disk excluding seek time
t_s: average time (in seconds) to read a block of 4 kilobytes from disk including seek time
tt: average time (in seconds) to transfer a byte from one machine to another

In the local index, the query $\vec{q_i}$ is sent to all processors, which implies that each processor has to execute a query whose length is given by q_i. In the lexicographical index, the query is broken in subqueries that are sent to the processors that hold the relative inverted lists, which implies that each processor j has to execute only a subquery $\vec{q}_{i,j}$ whose length is given by $q_{i,j}$. In the random index, the query is broken in subqueries that are sent to the processors holding blocks of their inverted lists, which implies that each processor j has to execute only a subquery $\vec{q}_{i,j}$.

When a processor j receives a query $\vec{q_i}$ (or subquery $\vec{q}_{i,j}$), it reads from disk the inverted lists relative to their terms and processes such lists to generate the local answer set. Considering that the inverted list for term k has a size given by $g\ell_{k,j}$, the number of blocks read from disk is given by $\sum_{k \in \vec{q_i}} g\ell_{k,j}$.

The variable $h_{i,j}$ counts the number of pairs <document, similarity> inserted or accumulated in the set of accumulators (represented by a hash table) during ranking evaluation at the various processors. The variables $c_{i,j}$ and $cs_{i,j}$ count the number of operations (comparison and swap) executed during ranking evaluation at the various processors. The variable $a_{i,j}$ counts the number of pairs <document, similarity> from the top of the ranking which are returned as local answer set by processor j for query $\vec{q_i}$. The variable c_{i,a_i} counts the number of operations (comparison and swap) during merging of the local answer sets at the broker. The parameters f_1 and f_2 are used to determinate the number of documents to be sent back to the broker. The parameter b is the number of bytes of a pair <document, similarity>. The remaining parameters relate to system time measures.

4.2 Analysis of Query Processing Time

We distinguish five main phases during the distributed query processing: (1) reading of inverted lists from disk; (2) accumulation of document weights using the vector space model; (3) ranking of the local answer set; (4) transference of the local answer sets to the broker; (5) merging of the local answer sets at the broker. Phase 1 (F1) to phase 4 (F4) are executed by the processors while phase 5 (F5) involves only the broker. As we consider a concurrent query service scheme, these five phases are overlapped and interleaved during the execution of the various queries.

In the local index, each processor must execute the whole query, which implies that the processing of a query $\vec{q}_i$ lasts at processor j for a time $t_{i,j}$, given by:

$$t_{i,j} = \begin{array}{ll} q_i \times t_s + \sum_{k \in \vec{q}_i} \ell\ell_{k,j} \times t_r + & (F1) \\ h_{i,j} \times t_h + & (F2) \\ c_{i,j} \times t_c + & (F3) \\ f_1 \times b \times tt & (F4) \end{array} \quad (1)$$

In the lexicographical index, the broker first determines which processors hold inverted lists of the query terms, breaks the query into subqueries and sends the subqueries to the respective processors. The time $t_{i,j}$ to process subquery $\vec{q}_{i,j}$ at processor j can be modeled by:

$$t_{i,j} = \begin{array}{ll} q_{i,j} \times t_s + \sum_{k \in \vec{q}_{i,j}} g\ell_k \times t_r + & (F1) \\ h_{i,j} \times t_h + & (F2) \\ cs_{i,j} \times t_c + & (F3) \\ p \times f_2 \times b \times tt & (F4) \end{array} \quad (2)$$

In the random index, the broker determines which processors hold blocks of the inverted lists relative to the query terms, breaks the query into subqueries and sends them to the respective processors. Only the time for F1 changes, given by:

$$q_{i,j} \times t_s + \sum_{k \in \vec{q}_{i,j}} g\ell_{k,j} \times t_r \quad (F1) \quad (3)$$

For the local index, the time $t_{i,b}$ to merge the local answer sets of query $\vec{q}_i$ is given by:

$$t_{i,b} = c_{i,a_i} \times t_c \quad (F5) \quad (4)$$

For merging the local answer sets, we use the multiway merge [18] algorithm, whose complexity is $O(f_1 . \log(j))$.

For both global indexes (lexicographical and random), the time $t_{i,b}$ to merge the local answer sets of query $\vec{q}_i$ must include the addition of weights of documents present in more than one local answer set, being given by:

$$t_{i,b} = a_i \times t_h + c_{i,a_i} \times t_c \quad (F5) \quad (5)$$

For adding the weights of documents, we insert each of the pairs <document, similarity> in a set of accumulators (represented by a hash table), with a complexity of $O(a_i)$; for sorting the total number of documents, we use the quicksort algorithm, whose complexity is $O(a_i . \log(a_i))$.

In the distributed query processing of a query $\vec{q}_i$, some processors finish the execution of F4 before others. These processors can then start immediately the execution of the next query, instead of waiting for the conclusion of F5 for the query $\vec{q}_i$. To capture such behavior, we employed a simulator to replicate the execution for a batch of queries of F1 through F4 in each processor, and estimate the processing time of each query in each of the processors in the system, according to Eq.(1), Eq.(2) and Eq.(3) for the local, lexicographical and random indexes, respectively. Also, the simulator processes and times F5 in the broker for each query $\vec{q}_i$, using Eq.(4) and Eq.(5) for the local, and lexicographical and random indexes, respectively.

We applied other simulator to estimate contention ct_i before service, starting time ts_i and finishing time tf_i of query $\vec{q}_i$ in any of the processors, and contention $ct_{i,b}$ in the broker, for varying request rates. In this way, for all the index partitioning strategies, the final processing time t_i of a query $\vec{q}_i$ is given by:

$$t_i = ct_i + max_{j=1}^{p}(tf_{i,j}) - min_{j=1}^{p}(ts_{i,j}) + ct_{i,b} + t_{i,b}$$

where $max_{j=1}^{p}(tf_{i,j})$ is the maximum finishing time of query $\vec{q}_i$ at processors $j = 1$ to p, and $min_{j=1}^{p}(ts_{i,j})$ is the minimum starting time of query $\vec{q}_i$ at processors $j = 1$ to p.

To estimate the variables involved in each of the phases at the processors, the simulator replicates the execution of a batch of queries in each processor and counts: the number of terms of the query (or subquery) and the number of blocks read in F1; the number of pairs <document, similarity> inserted or accumulated in the set of accumulators in F2; the number of documents ranked and the number of operations of comparison or swap for ranking documents in F3; and the number of bytes transferred to the broker in F4. Also, the simulator processes the merging of the local answer sets for a batch of queries to count the size of the local answer sets (in lexicographical and random indexes) and the number of operations of comparison and swap done for generating the final ranking of documents in F5 at the broker.

4.3 Experimental Setup

The machine we used in experiments is an AMD-K6-2 with a 500 MHz processor, 256 megabytes of main memory, 30 gigabytes IDE hard disk, and running Linux operating system version 2.4.7-10.

The data comprise 20 gigabytes of Web pages collected by the TodoBR [16] search engine. Real lengths of the inverted lists in TodoBR are used. The query set is composed by 100, 256 queries of a partial log of queries submitted to TodoBR. In this log, there is a total of 37,450 unique queries, and 23,751 unique terms.

4.4 Validation of the Analytical Model

For predicting I/O time for accessing data on disk, we made experiments using a raw device, which can be bound to an existing block device (e.g. a disk) and be used to perform raw I/O with that existing block device. Such raw I/O bypasses the caching that is normally associated with block devices. Hence a raw device offers a more direct route to the physical device and allows an application more control over the timing of I/O to that physical device. The system time parameters used in our analysis, which we have shown to be valid in our test machine, are given in Table 1.

We firstly measured the service time to random data blocks, for a large number of requests, and then estimated the mean of these times. Secondly, we measured the service time of sequential data blocks, and again estimated the mean of these times. The first mean t_s includes seek time, as it relates to randomly accessed data blocks. The second mean t_r excludes seek time, because it relates to sequentially accessed data blocks. In this way, we consider t_s while reading the first block of an inverted list and t_r while reading the subsequent blocks of the same inverted list.

In order of inferring CPU time for retrieving documents,

System Time Parameters	
Disk read (including seek time)	1.02192×10^{-2} secs per block of 4 kilobytes
Disk read (excluding seek time)	2.99654×10^{-4} secs per block of 4 kilobytes
Insertion in the set of accumulators	8.03982×10^{-7} secs per pair <document, similarity>
Operation of floating point division	1.6892×10^{-7} secs
Operation of comparison or swap	5.82213×10^{-8} secs per pair <document, similarity>
Network transfer time	2.4×10^{-7} secs per byte

Table 1: System time parameters in our test machine.

we measured the time to insert a number of pairs <document, similarity> in the set of accumulators represented by a hash table, and then estimated the average time t_h to insert one pair in the hash. In fact, we timed this insertion procedure for an increasing number of pairs and confirmed that our predictions are accurate ones. For predicting CPU time for ranking documents, we timed the sorting of a number of pairs <document, similarity> and estimated the average time t_c of an operation of comparison or swap during sorting. Again, we repeated this sorting procedure for an increasing number of pairs to certify the precision of our estimations.

Once we had derived system time parameters, in order of validating F1 to F3 of our analytical model, we timed the execution of these phases for 100 batches of 50 queries in a single processor. Each batch were executed 10 times, which proved to be a sufficient number of runs with an estimated coefficient of variation of 1.02% on average.

We also evaluated the execution of F1 to F3 for these same 100 batches of 50 queries using our analytical model. The results are shown in Figure 3. As can be seen, agreement between the real and the analytical processing times is rather good, with an estimated relative deviation of 3.12%.

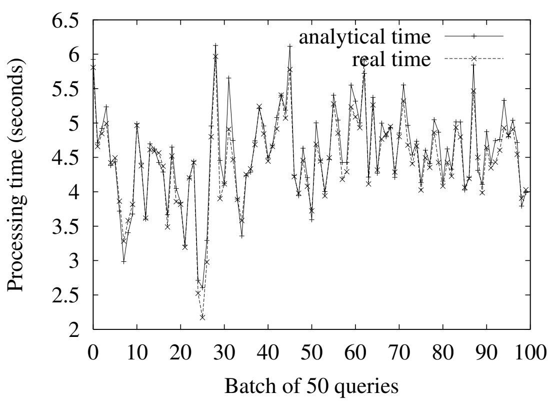

Figure 3: Comparison between real and analytical processing times for 100 batches of 50 queries.

Regarding F5, it is entirely analogous to F3 in the local index, and a combination of F2 and F3 in the global index. In this way, our prediction is fairly accurate, because it worked for F2 and F3. Regarding F4, we estimated a value for network transfer time based on a third of the nominal value of our network, in order to take into consideration the overheads due to the operating system and the communication protocols which affect the transmission of data.

5. RESULTS

The random global, lexicographical global, and local indexes are compared in the aspects of concurrency and parallelism, disk seek, load balance, size of inverted lists, fault tolerance, and system configuration, as presented in Table 2.

Experimental results show the practical impact of the three index partitioning strategies on the performance of our distributed information retrieval system. We employ a simulator to read a log of 100, 256 (a hundred thousand and two hundred fifty six) queries submitted to the TodoBR search engine, replicate its execution and generate a trace with the processing time, estimated by the analytical model, of each query in each of the processors in the system.

For analyzing the performance of the system for varying request rates, we apply other simulator to read the timing trace and report, among other information, the average response time for the system to send an answer, the throughput of the system, the average cost by active processor, and the average load imbalance by active processor. The cost is the processing time of a processor, and the load imbalance is the difference between the processing time of the processors and the average processing time by active processor.

Figure 4 presents the characteristics of the inverted lists of the TodoBR log used in the experiments. Regarding the size of the inverted lists assigned to each processor, in the local index and random global index inverted lists are smaller because in the former they contain only the documents from the local subcollection, and in the latter only the documents from the blocks of inverted lists assigned to the processor. On the other hand, in the lexicographical global index the inverted lists assigned to each processor are larger, because they contain documents from the whole text database collection.

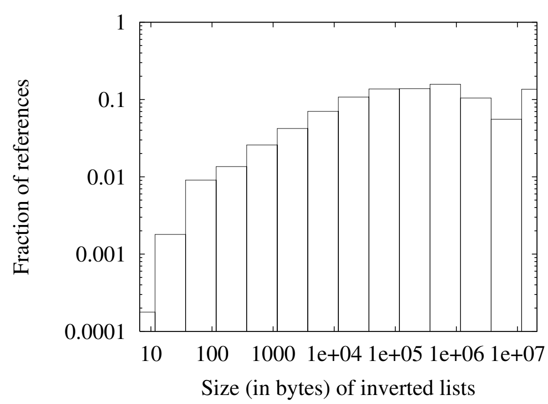

Figure 4: Popularity versus size for inverted lists, relative to a TodoBR log of roughly 100 thousand queries.

LI	LGI	RGI
High parallelism	High concurrency	Both parallelism and concurrency
More disk seeks	Less disk seeks	Less disk seeks
Better load balance	Worse load balance	Better load balance
Smaller inverted lists	Larger inverted lists	Smaller inverted lists
Reasonable fault tolerance	Low fault tolerance	High fault tolerance
Reasonable system configuration	Harder system configuration	Easier system configuration

Table 2: Comparison between the local, lexicographical global, and random global index partitioning strategies.

Regarding concurrency and parallelism, in the local index all processors are devoted to the execution of a single query, thus providing high parallelism. In the lexicographical global index, not all processors might be involved with the processing of a single query (e.g., when the number of processors is larger than the number of query terms, or when many query terms are stored in a single processor releasing the others), thus allowing high concurrency. The random global index combines both parallelism and concurrency. Parallelism is allowed because the blocks of an inverted list are spread among processors, which implies that more than one processor might be involved with the processing of a same term mentioned in a query. Concurrency is allowed because if only a small number of processors happen to hold all the blocks of the terms of a given query, then those processors are able to execute that query without need to cooperate with the others. As a result, more than one query might be processed simultaneously.

Regarding disk seek, in the local index retrievals require more disk seeking operations, because each processor receives all query terms. In the lexicographical global index and random global index retrievals require less disk seeking operations, because the processors do not necessarily receive all query terms.

Figure 5 shows the response time as a function of the request rate for 8 and 64 processors. The configurations employing 64 processors provided better response times and RGI outperformed LGI by 18% on average for the interval between 1 and 50 requests per second. As the number of request per second increases, and because the RGI organization achieves more parallelism, the workload of the broker increases, becoming a relevant part of the response time. Hence, the LGI becomes better.

Figure 6 shows the throughput as a function of the request rate for 8 and 64 processors. We observe that the LGI configurations achieved higher throughput rates, where we observe average gains of about 10% of LGI over RGI. Further, as expected, both RGI and LGI outperformed significatively LI, where the gains reached five times.

Regarding load balance, in the lexicographical global index the load balance level is worse than in the random and local index. The reason is that in the lexicographical global index, the terms in a query are sent only to the processors which store their inverted lists. This implies that the processor that holds the most frequent terms in the query is heavily loaded, while the processor that holds the least frequent query terms stays relatively idle. In the random global index, load balance is better, because fixed sized blocks of an inverted list are randomly distributed among processors. In this way, the probability distribution of blocks in the processors tends to be uniform, which provides a good load balance. In the local index, all terms of a query are sent to all processors and a good load balance level is always provided.

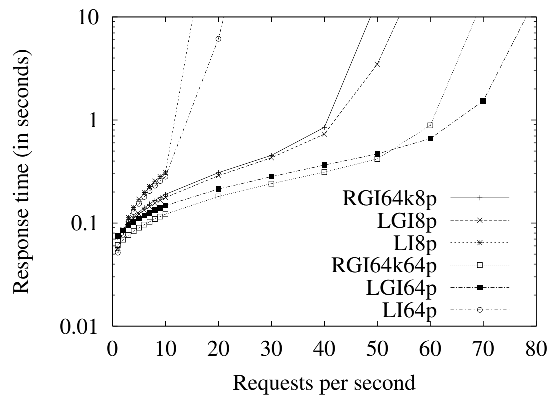

Figure 5: Average response time as a function of the request rate, for the random global index (RGI) with block size of 64 kilobytes, lexicographical global index (LGI) and local index (LI), with 8 and 64 processors.

Figure 7 shows the load imbalance as a function of the number of processors, where we can see that RGI reduces the LGI load imbalance by about half. We explain these gains by verifying how the different index partitioning strategies affect the load balance among the various processors. LGI is well-known for its poor load balancing, since the processing is quite localized and the sizes of the lists merged for answering a given query are usually quite different.

Another factor that helps reducing the load imbalance is that the average number of processors per query employed by RGI is consistently greater than the number of processors employed by LGI, as shown in Table 3. A better usage of computational resources, as well as the distribution of list segments among processors explain the better load balancing associated with RGI. LI, as expected, presents the lowest load imbalance. The better results provided by RGI are also explained by the average processing cost per processor, as shown in Figure 8, where we observe that not only the cost (in time) for 64 processors is 20% smaller than the other strategies, but also that the gain increases with the number of processors. This last observation is a good indication of the better scalability of RGI, making it suitable for large scale systems.

Finally, we analyzed the trade-offs in terms of the block size. As expected, the average cost per processor increases with the block size and the number of processors used per query decreases as the block size increases, as shown in Figures 9 and 10.

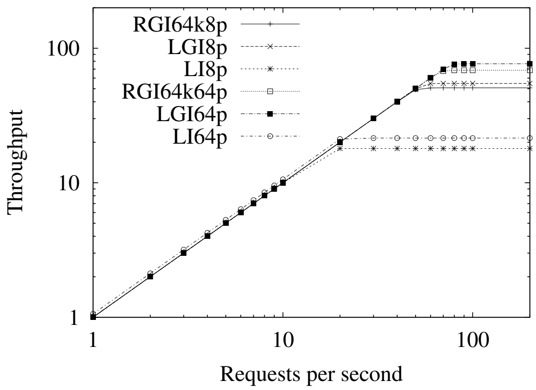

Figure 6: Throughput obtained by the system as a function of the request rate, for the random global index (RGI) with block size of 64 kilobytes, lexicographical global index (LGI) and local index (LI), with 8 and 64 processors.

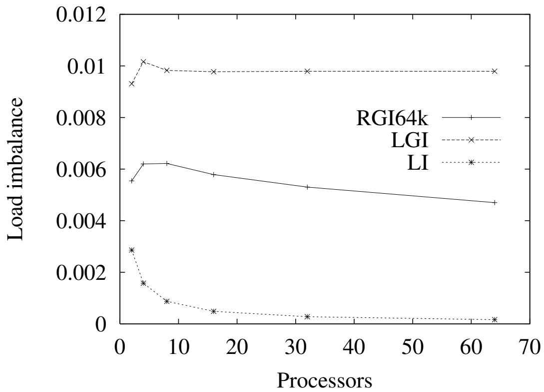

Figure 7: Average load imbalance by active processor as a function of the number of processors in the system, for the random global index (RGI) with block size of 64 kilobytes, lexicographical global index (LGI) and local index (LI).

Regarding fault tolerance, the random global index and local index are more resilient to failures than the lexicographical global index. In the lexicographical global index, when a processor that holds a set of terms fails, the queries that refer to terms in that set cannot be answered. On the other hand, a failure of a processor in the random global index represents the loss of only a subset of documents present in the blocks held by that processor. Also, in the local index a failure does not prevent any query from being answered, though the final answer set might not contain all the relevant documents in the collection.

System reconfiguration is easier with the random global index than with the local and lexicographical global index. If more processors are added to a system using local index, some documents have to be re-distributed across processors and inverted locally. With the lexicographical global index, the addition of more processors requires the re-distribution of global inverted lists among processors. On the other hand, with a random allocation approach only a fraction of random selected blocks have to be moved to the new processors, in order to keep the average load balanced across the processors.

Number of processors in the system	Ratio between RGI and LGI of the average n. of processors used per query
2	1.0362
4	1.0588
8	1.1019
16	1.1186
32	1.1276
64	1.1355

Table 3: Ratio between the random global index (RGI) with block size of 64 kilobytes and lexicographical global index (LGI) of the average number of processors used per query.

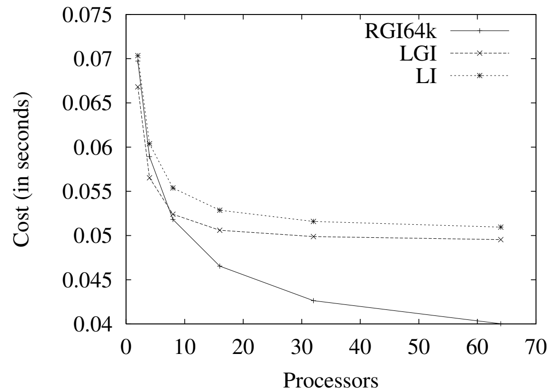

Figure 8: Average cost (in time) by active processor as a function of the number of processors in the system, for the random global index (RGI) with block size of 64 kilobytes, lexicographical global index (LGI) and local index (LI).

6. CONCLUSIONS AND FUTURE WORK

A lexicographically partitioned global index is a quasi-random order for the inverted lists. We could hash the vocabulary to have a real random order, but still, parallelism does not increase too much because we have very long lists that are used partially or belong to words that are never queried. In this paper we proposed a technique that allows a full random order by splitting the inverted lists in blocks and randomizing its processor allocation. Our simulation results show that this technique outperforms the lexicographical global index.

Further work includes a thorough experimental comparison, tuning our technique to find the optimal block size, and coupling it with other solutions to increase parallelism and load balance, such as using query frequencies or hierarchical indexes. Further work is also needed in better broker design, such that is not the bottleneck for large request rates. This can be achieved also exploiting parallelism.

7. REFERENCES

[1] C. S. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In *Proceedings of the 8th*

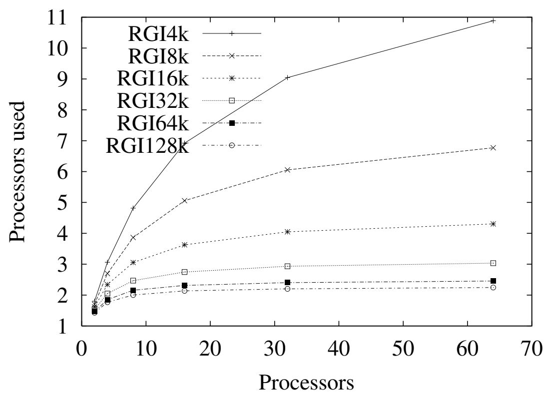

Figure 9: Average number of processors used per query as a function of the number of processors in the system, for the random global index (RGI) when varying the block size from 4 to 64 kilobytes.

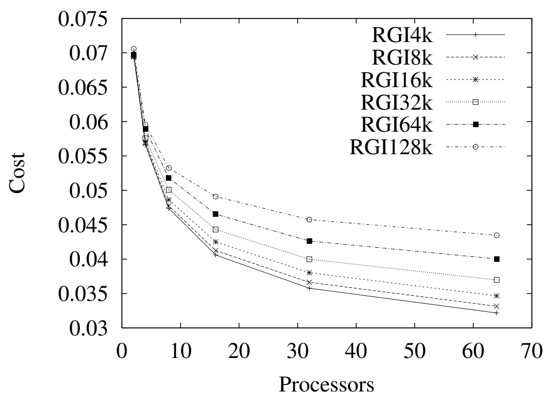

Figure 10: Average cost by active processor as a function of the number of processors in the system, for the random global index (RGI) when varying the block size from 4 to 64 kilobytes.

String Processing and Information Retrieval Symposium, pages 10–20, Laguna de San Rafael, Chile, 2001. IEEE Computer Society.

[2] R. Baeza-Yates and B. Ribeiro-Neto, editors. *Modern Information Retrieval.* ACM Press New York, Addison Wesley, 1999.

[3] S. Berson, R. Muntz, and W. Wong. Randomized data allocation for real-time disk i/o. In *Proceedings of the COMPCON'96*, pages 286–290, 1996.

[4] Y. Birk. Random raids with selective exploitation of redundancy for high performance video servers. In *Proceedings of the 7th International Workshop on Network and Operating Sytem Support for Digital Audio and Video (NOSSDAV'97)*, pages 13–23, St. Louis, MO, 1997.

[5] D. Harman. Overview of the third text retrieval conference. In D. Harman, editor, *Proceedings of the Third Text REtrieval Conference (TREC-3)*, pages 1–19, Gaithersburg, Maryland, U.S.A., 1994. NIST Special Publication 500-207.

[6] D. Hawking, N. Craswell, and P. Thistlewaite. Overview of TREC-7 very large collection track. In E.M. Voorhess and D.K.Harman, editors, *Proceedings of the Seventh Text Retrieval Conference*, pages 257–268, Gaithersburg, Maryland, U.S.A., 1998. NIST Special Publication 500-242.

[7] B. S. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, 1995.

[8] J. Korst. Random duplicated assignment: An alternative to striping in video servers. In *Proceedings of the ACM Multimedia 97*, pages 219–226, Seattle, WA, USA, 1997.

[9] A. MacFarlane, J. A. McCann, and S. E. Robertson. Parallel search using partitioned inverted files. In *Proceedings of the 7th International Symposium on String Processing and Information Retrieval*, pages 209–220, La Coruna, Spain, 2000. IEEE Computer Society.

[10] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[11] B. A. Ribeiro-Neto and R. A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the third ACM Conference on Digital Libraries*, pages 182–190, 1998.

[12] G. Salton. *The SMART retrieval system – Experiments in automatic document processing.* Prentice Hall Inc., Englewood Cliffs, NJ, 1971.

[13] G. Salton and C. Buckley. Term-weighting approaches in automatic retrieval. *Information Processing and Management*, 24(5):513–523, 1988.

[14] J. R. Santos, R. R. Muntz, and B. A. Ribeiro-Neto. Comparing random data allocation and data striping in multimedia servers. In *Proceedings of the ACM SIGMETRICS 2000*, pages 44–55, Santa Clara, California, USA, 2000.

[15] R. Tewari, R. Mukherjee, D. M. Dias, and H. M. Vin. Design and performance tradeoffs in clustered video servers. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems 1996 (ICMCS'96)*, pages 144–150, Tokyo, Japan, 1996.

[16] TodoBR. Main page: http://www.todobr.com.br.

[17] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 8–17, San Diego, California, U.S.A., 1993.

[18] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes - Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, Inc., 2ª edition, 1999.