Impact of the Query Model and System Settings on Performance of Distributed Inverted Indexes

Simon Jonassen and Svein Erik Bratsberg

Abstract

This paper presents an evaluation of three partitioning methods for distributed inverted indexes: local, global and hybrid indexing, combined with two generalized query models: conjunctive query model (AND) and disjunctive query model (OR). We performed simulations of various settings using a dictionary dump of the TREC GOV2 document collection and a subset of the Terabyte Track 05 query log. Our results indicate that, in situations when a conjunctive query model is used in combination with a high level of concurrency, the best performance and scalability are provided by global indexing with pipelined query execution. For other situations local indexing is a more advantageous method in terms of average query throughput, query wait time, system load and load imbalance.

Acknowledgements

Further details on the background review, the simulation model itself and the experimental results are available at [4]. The source code for the simulation model, test data, experiment configuration and result files can be obtained from [5].

1 Introduction

An *inverted index* maintained by a single node is considered to be an efficient approach to perform query-based search within a document collection [10]. For a distributed architecture, *index partitioning* is considered to be an efficient technique to handle a larger document collection or to reduce query latency by creating a number of non-overlapping indexes, or *partitions*, and by distributing them on different nodes in a cluster.

This paper was presented at the NIK-2009 conference; see http://www.nik.no/.

An inverted index can be partitioned either by terms (*global inverted indexing*) or by documents *local inverted indexing*. Alternatively, each inverted list can be distributed among the nodes block by block using a hash function. The latter method is known as *hybrid indexing*. Finally, a method known as *pipelined architecture* applies a sequential sub-query execution on a global inverted index in order to minimize network load and computation load on the *receiving* (receptionist) node.

The main objective of this paper is to perform a quantitative re-evaluation of all four methods and determine at which conditions each of the methods is more advantageous. As the methods presented in this paper are not new, the contribution of this paper is an evaluation of these along with two general query models and varied system settings. Simulation was chosen because it allows to change system parameters such as network and disk bandwidth, disk access time and CPU performance factor.

This paper is organized as follows. Section 2 introduces partitioning methods and provides a short summary of their major pros and cons. Section 3 gives a short overview over previously published papers, assumptions and results. Section 4 outlines the simulation model, algorithms implemented, system variables and input data. Some of the experimental results are listed and evaluated against previously published results in Section 5. Final conclusions and proposals for a further research follow in Section 6.

2 Partitioning Methods for an Inverted Index

As the methods we evaluate in this work are not novel, this section presents only a short summary of main advantages presented in Table 1. A full description of these can be found in [4]. General differences between the methods to be studied can be explained as follows:

Local Indexing (LI) - With LI the whole document collection is divided into a number partitions assigned to different nodes. Then each node processes its documents and stores a local inverted index.

Global Indexing (GI) - With GI a single inverted index for the whole document collection is constructed in the first place, thereafter the index is distributed term-wise between the nodes.

Pipelined Architecture for GI (PI) - In order to minimize the amount of data to be processed and transferred during query processing, [8] proposed to create a query bundle and route it based on the least highest term frequency at each node and post-process the final results on the last node.

Hybrid Indexing (HI) - In order to combine advantages of GI and LI, [11] proposed a technique called hybrid indexing. The main difference from GI is that each posting list is divided into *chunks*, or blocks, which are distributed according to term ID XOR chunk ID.

Table 1: Summary of characteristics. In this example execution of a single query involves q terms of an index distributed over k nodes and requires to return K top scored results. Expected total length of an inverted list for a term t is l.

	LI	GI	PI	HI
index construction/update	easy	moderate/difficult	same as GI	difficult
querying	easy	moderate	moderate/difficult	difficult
pre-processing required	no/yes	yes (lookup)	yes (route)	yes (lookup)
number of sub-queries	k	q	1 (query bundle)	1 - k
total avg. number of disk seeks	kq	q	q	1 - kq
worst case avg. number of disk seeks on each node	q	q	q	q
expected length of an inverted list stored on a node	l/k	l	l	l/k
query concurrency support	moderate	good	moderate	moderate
sub-query processing requirements	moderate/high	low/moderate	low	moderate/high
number of partial results to be returned to the receiving node	K	all	all	all
post-processing requirements	low/moderate	high	low	high
possibility of disk bottleneck	high	low	moderate	moderate
possibility of receiving node bottleneck	low	high	moderate	high
possibility of network bottleneck	low	high	high/moderate	high

3 Related Work

During the last 20 years a number of publications elaborate on the comparison between local and global indexing, but the results from various studies tend to be inconsistent.

A number of techniques designed for a multi-disk system and a couple of optimizations for these have been explained and evaluated in [12], where a small simulator with a synthetic document collection and synthetic Boolean conjunctive queries were used to perform experiments under varied system settings, such as disk access time and network bandwidth.

Local and global indexing in the original form were first presented and compared using a simulation model in [3]. The simulation model described a shared-everything system and used also a synthetic document collection and a synthetic query set.

In [9] a simple analytical model together with a small simulator using the TREC3 document collection and 50 real queries were used to evaluate performance of local and global indexing on a shared-nothing system with a varied number of nodes. It also presented results from two interesting system setups referred as 'fast disk' and 'slow network' indicating that each of the indexing schemes may be advantageous depending on the system settings. However, in contrast to the work in [3, 12], this work had no evaluation of varied concurrency level (i.e. number of queries executing concurrently), and for some of the experiments the number of nodes (64) exceeded the size of the query set (50).

A later contribution to this work was presented in [2], which used a real implementation, 50 real queries and a synthetic query set consisting of 2000 queries. However, the only varied system parameter evaluated in this paper was the number of processors (nodes) and other system settings were fixed.

The work presented in [6] evaluated a real implementation using two subsets of 100GB VLC2 document collection and a number of queries obtained from the TREC-7 topic descriptions. Contributions of the paper included a probabilistic query model instead of a vector or Boolean model. It also evaluated different distribution schemes for local indexing. However, both the document collection and the query set sizes were equal or smaller than those used in [9]. Surprisingly enough, [6] concluded with the results opposite to those listed in [9].

The work presented in [1] listed a number of advantages of using local indexing on a shared-nothing system, such as good disk and CPU utilization levels, lower imbalance and avoidance of a network or receptionist bottleneck, proved by a number of experiments with TodoBR search engine using a 16GB large index and 20000 real queries. However, this work contained no corresponding evaluation of global indexing and again none of the system parameters have been varied.

An experimental study aimed to show the advantage of hybrid indexing based on a real implementation, 100GB TREC-9/10 and a query set consisting of 2000 real queries was provided in [13]. Issues such as change in performance with an increasing concurrency level and load balancing have been evaluated in this work, however it does not explain which query model or what kind of queries have been used. Additionally, some of the observations listed in the paper were quite surprising while the reasons for these were not explained.

A pipelined architecture for global indexing was first presented in [8], which compared this method against conventional implementations of local and global indexing with changing collection size and number of processing nodes using a modified version of the Zettair search engine, 426GB of the TREC GOV2 collection and a pseudo-realistic query set.

The major problem of the pipelined architecture observed in [8] is load balancing. In order to improve load balancing, a re-evaluation of a modified

method was presented in [7]. The approach and the system setup were similar to [7], while the queries were submitted in batches and the number of nodes was fixed.

Directions for This Work

Both the approach and the experiments to be presented later in this paper are quite similar to [12] and [3]. However, in contrast to [12] and [3], this work evaluates partitioning methods for a shared-nothing multi-node system. In addition, the experimental model is based on a real document collection and a real query log.

In contrast to [9] this work evaluates different concurrency levels. In contrast to later publications this work performs a re-evaluation of all three partitioning methods and a pipelined architecture under varied system settings. Additionally, it evaluates two generalized query models in order to show that the final system performance is affected by the choice of the query model. All the publications presented so far perform their evaluation only on a single query model.

4 Simulation Model

The source code in Java and implementation details of the simulation model are provided in [4] and [5]. The simulation model for the disk and network access is quite simple and does not model any advanced features such as caching and prefetching. The system performance metrics used in the model, such as processing times, disk access characteristics and network delay, were gathered either from a number of micro-benchmarks performed on a Intel Pentium IV, 3.0GHz, 1GB RAM running Java 1.6 or from datasheets for Seagate Barracuda Hard-Drives.

The document collection is simulated using a dumped dictionary of the TREC GOV2 document collection containing 32 801 629 unique terms contained in 25 205 179 documents, filtered through a dictionary for the Terabyte Track 05 query set containing 50 000 queries. As a result, the query set contains 31 220 unique terms matching within the document collection, and 3 912 terms without any match. None of the queries of the original query set is modified, but only a small number of queries are used.

Instead of choosing a query to document similarity model, two generalized models for term-matching have been considered. The first one represents a query a conjunction of terms, while the second one represents a query as disjunction of terms. The occurrence probability of any term is assumed to be independent of any other term occurrences. In this case, the joint probability of a number of terms using the conjunctive query model (AND) is a product of their individual probabilities. For the disjunctive query model (OR), the total probability is one minus a product of improbabilities of every single term. A consequence of this

assumption is that the estimated number of documents matching the phrase is either too high or too low compared to the real world.

Implemented Algorithms

Using the simulation model four algorithms were implemented, LI AND/OR, GI AND/OR, PI AND/OR and HI OR. Algorithm details as well as a corresponding mathematical model can be found in [4] pp. 65 - 72. A simplified description of LI AND/OR algorithm is illustrated by Algorithm 1 - 2. The implementation of GI AND/OR differs from PI AND/OR in additional tasks during query processing, while processing of sub-queries is simplified. The implementation of PI AND/OR is just a sequential version of GI AND/OR, while HI OR is quite similar to GI OR with a difference supporting the fact that each inverted list is now distributed across all of the nodes.

Note that HI AND was not implemented since corresponding postings for different terms are mapped to different nodes when HI is applied, therefore an implementation of HI AND would require to return complete inverted lists to the receiving node. Finally, due to computational complexity of HI OR, it has been simulated only on a collection 10 times smaller than the original one.

1 Receive a query from the gateway node;
2 Allocate and initialize all necessary data structures;
3 Generate and schedule the sub-queries for this query;
4 Initialize an accumulator list;
5 Wait for partial results to be returned;
6 Merge the partial results in parallel, determine top-scored results;
7 Transfer the results back to the gateway node;
8 Free memory and terminate;

Algorithm 1: LI query processing, algorithm details

5 Experimental Results

The results presented in this section are based on a simulated system with 4 and 8 nodes. This is consistent with [12] and [2] which provide results for up to 4 nodes, and [6], [8] and [7] which provide results for up to 8 nodes. Only [9] presents results for a system with up to 64 nodes.

All the metric result data presented in this section were gathered from 50000 millisecond long query runs while all the explanations given are based on 5000 millisecond long trace runs.

Impact of the Query Model

Table 2 presents the main performance metrics of LI, GI and PI in combination with AND and OR queries. From the results presented LI provides the best

```
1  Receive a sub-query from the query node;
2  Perform a dictionary look-up for the sub-query terms if required;
3  if at least one term is missing then
4      if system is in AND-mode then
5          Transfer 0 results to the receiving node;
6          Free memory and terminate;
7      else
8          Eliminate non-existing terms;
9  Allocate and initialize an accumulator list;
10 for sub-query terms ordered by increasing f_t do
11     Fetch the inverted list for a term;
12     Merge the inverted list into the accumulator list;
13 Determine and sort top-scored results;
14 Transfer the results back to the receiving node;
15 Free memory and terminate;
```

Algorithm 2: LI sub-query processing, algorithm details

performance when a disjunctive query model is used. In this case it provides both the highest query per second (QPS) rate, shortest query response time, and lowest imbalance and average load for both network and memory. The number of disk seeks, however, is highest for LI and the disk utilization is 100%. The latter observation supports the scepticism about a potential disk bottleneck when LI is used.

On the other hand, GI OR shows the worst performance for every parameter except from the disk load, which is only 56%, but the imbalance is about 35%. Also the network load for GI OR is 260 times higher than for LI OR, while the imbalance is quite low.

PI shows an even longer query response time. The reason for this is a way too high network imbalance in addition to a high network load, since the amount of data transferred between successive stages only increases. But the CPU load for PI is even lower compared to LI and GI, since CPU is involved only in calculation of new data, rather than re-processing of old data.

The situation is very interesting for the conjunctive queries. In this case the relationship between GI and LI is the same, while PI AND provides both the highest QPS rate and the lowest query response time. The reason for this is a great reduction in the amount of data to be transferred in later pipeline stages, and as a result a very low network load (only 0.5%, which is a reduction by 98%). Since the bottleneck for GI lies not in the amount of data to be transferred or processed later, but in the amount of data to be fetched from disk, the performance improvement of GI is very small.

Table 2: Results summary of the baseline experiments (50000ms, 4 nodes, max. number of concurrent queries 12)

	LI or	GI or	PI or	LI and	GI and	PI and
QPS	4.56605	2.98764	3.64826	4.62639	3.68944	4.73072
Avg. query resp. time	2589.2	3102.4	3117.1	2542.9	2987.7	2452.7
Avg. CPU load	0.26709	0.28589	0.26113	0.16420	0.19519	0.16741
Tot. avg. CPU imb.	1.00000	1.26076	1.23165	1.00055	1.36631	1.17353
Avg. disk load	1.00000	0.55546	0.66923	1.00000	0.66284	0.82174
Tot. avg. disk imb.	1.00000	1.35075	1.18793	1.00000	1.32525	1.15859
Avg. eth. load	0.00151	0.39689	0.36311	0.00134	0.18873	0.00597
Tot. avg. eth. imb.	1.00000	1.09310	1.22522	1.00746	1.36793	1.17923
Avg. mem. load	0.17067	0.41655	0.28384	0.16675	0.46903	0.27931
Tot. avg. mem. imb.	1.00205	1.21476	1.19955	1.00222	1.26860	1.23515
Tot. disk seek	5060	1283	1561	5084	1568	1977

Impact of the Number of Nodes and Query Concurrency Level

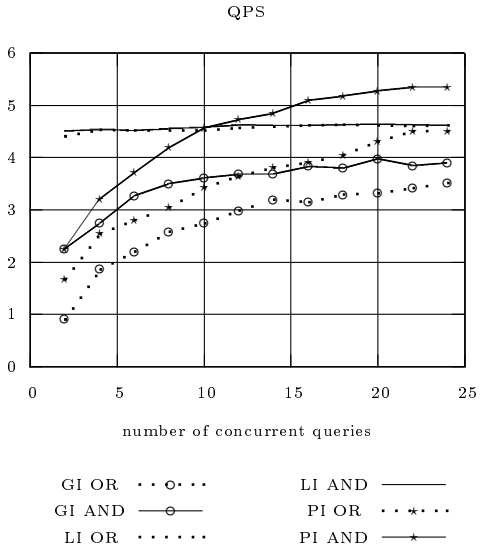
In order to perform an evaluation similar to [12] and [3], effects of a varied number of nodes and concurrency level have been studied. As Figure 1 shows LI does not scale at all, since LI encounters a disk bottleneck even at a low query concurrency level. Both GI OR and PI OR, on the other hand, improve their performance gradually, while they do not provide a QPS rate higher than the one provided by LI. However, PI AND outperforms GI in terms of the QPS rate at concurrency levels higher than 10.

The results are even more interesting for a system with 8 nodes. As Figure 2 shows, PI outperforms LI in terms of the QPS rate at a concurrency level higher than 9 for AND-queries and higher than 20 for OR queries. PI provides better results also in terms of the average query response time.

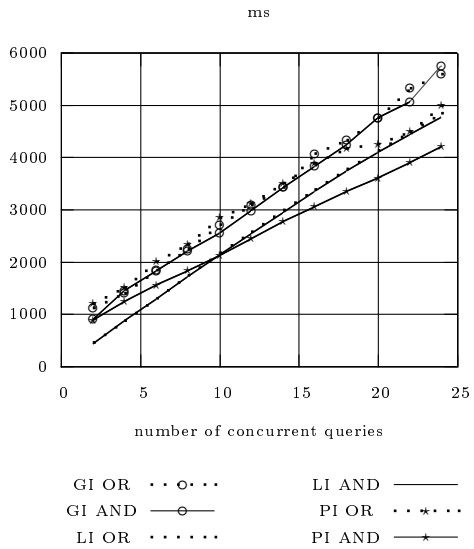
Impact of CPU, Network and Disk Configuration

In order to evaluate how do CPU, network and disk parameters affect performance of the indexing schemes a large number of experiments have been conducted. Results and details for these can be found in [4]. In general it was observed a linear decrease in the CPU utilization when the number of CPUs increases, but as CPU was not considered as a critical resource, neither the QPS rate nor the average query response times had changed. Simulations with a faster CPU indicated no significant change in the relationship between the partitioning schemes.

For a network with 10 and 100 times lower bandwidth the only observed change for LI was an increase in the average network load, while it was still below the full capacity even for a 100 times slower network. In contrast to LI, GI failed completely when the network got overloaded (though the average load value was not 1.0 due to the imbalance). PL OR performed close to LI OR with only a small

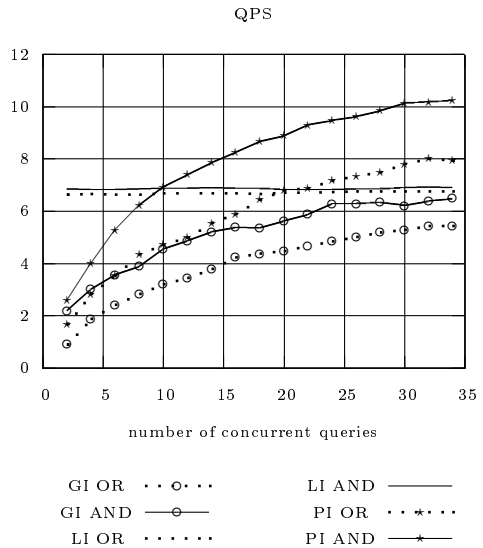


(a) Average QPS

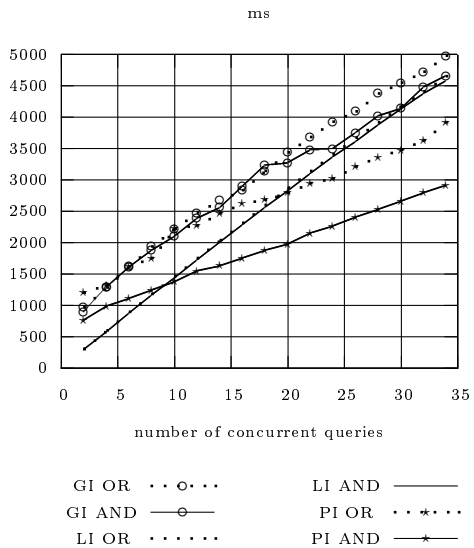


(b) Average query response time

Figure 1: Varied concurrency level using 4 nodes



(a) Average QPS



(b) Average query response time

Figure 2: Varied concurrency level using 8 nodes

difference between the QPS rate and the average query wait time.

From the experiments with different numbers of disks on each node, LI is more advantageous for a greater number of disks as the disk-access bottleneck becomes resolved. However, for 4 disks per node, the CPU load became about 1.0 when LI OR was used. Despite to this fact it was observed a super-linear increase in both the QPS rate and the query response time for LI caused by a better utilization of the other resources.

For different disk characteristics, such as shorter access times and higher bandwidth, LI was demonstrated to be more advantageous. These result could also be explained by resolved disk bottleneck, as it was mentioned earlier.

Evaluation of Hybrid Indexing

A number of the experiments aimed to compare HI OR against the other methods have been executed for a collection size of 250 000 documents (100 times smaller than the original one). From the results, HI provides a three times lower QPS rate and two times longer average query response time compared to GI.

Comparison to Previous Results

Local Indexing: From our results LI provides the best performance on disjunctive queries in terms of shorter disk and network access times, memory wait times and post-processing times. These results are similar to [3] with a difference in the fact that a good disk utilization mentioned in [3] is actually a disk bottleneck in some of the performed experiments. Another difference from [3] as well as [6] is that LI does not scale with increasing concurrency level.

Global Indexing: Some of the observations from [9] have been confirmed by our results, while the overall conclusion (that GI provides the best performance) was not confirmed. As [2] mentions that the implemented solution used a filtering technique, its overall conclusion is plausible as the results presented in our work show that concurrency and reduction in the data volume are the keys to a superior performance provided by GI. It was also confirmed that post-processing is a performance killer for GI, just as it was concluded in [6], but it can actually be omitted by reduction of the data volume with help of a more restricted query model or a filtering technique. Finally, our results show that GI (both GI AND and GI OR) provides a better scalability with the number of concurrent queries compared to LI.

Hybrid Indexing: Hybrid indexing did not show nearly the same results as in [13], which supports the scepticism to this method expressed in the other papers.

Pipelined Scheduling for Local Indexing: PI was demonstrated to be efficient when used together with a conjunctive query model, which provides better results than those presented in [8] and [7]. However, it must be kept in mind that the simulation model applied was much more optimistic with respect to the term correlation than a real implementation.

6 Conclusions and Further Work

A part of our results mainly confirms the previously published results - under normal settings local indexing is a more advantageous method in terms of the average query throughput, query wait time, system load and load imbalance. The novel results obtained in our work show that for higher concurrency levels the best performance is provided by global indexing with pipelined query evaluation when a conjunctive query model is used. It has two major points. First, our results demonstrate the impact of the query model and concurrency level on the performance of a distributed inverted index. Second, the conjunctive query model is the one applied by the most of the large scale search-engines, thus there is a potential for applying global indexing on these. Finally, we have also confirmed the scepticism about hybrid indexing.

In future, the simulation model should be focused on a more accurate performance model for network and disk access, more efficient posting list mapping schemes, simulation of filtering and caching methods for query processing. Finally, to be realistic, future results should be based on a system with a larger number of nodes.

References

[1] Claudine Badue, Ramurti Barbosa, Paulo Golgher, Berthier Ribeiro-Neto, and Nivio Ziviani. Basic issues on the processing of web queries. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 577–578, New York, NY, USA, 2005. ACM Press.

[2] Claudine Santos Badue, Ricardo A. Baeza-Yates, Berthier A. Ribeiro-Neto, and Nivio Ziviani. Distributed query processing using partitioned inverted files. In *SPIRE*, pages 10–20, 2001.

[3] Byeong-Soo Jeong and Edward Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):142–153, 1995.

[4] Simon Jonassen. Distributed Inverted Indexes. `http://daim.idi.ntnu.no/show.php?type=masteroppgave&id=4197`, 2008.

[5] Simon Jonassen. Source files for 'Distributed Inverted Indexes'. `http://daim.idi.ntnu.no/show.php?type=vedlegg&id=4197`, 2008.

[6] A. MacFarlane, J. A. McCann, and S. E. Robertson. Parallel search using partitioned inverted files. In *SPIRE '00: Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, page 209, Washington, DC, USA, 2000. IEEE Computer Society.

[7] Alistair Moffat, William Webber, and Justin Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 348–355, New York, NY, USA, 2006. ACM Press.

[8] Alistair Moffat, William Webber, Justin Zobel, and Ricardo Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 10(3):205–231, 2007.

[9] Berthier A. Ribeiro-Neto and Ramurti A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *DL '98: Proceedings of the third ACM conference on Digital libraries*, pages 182–190, New York, NY, USA, 1998. ACM Press.

[10] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[11] Ohm Sornil. *Parallel inverted index for large-scale, dynamic digital libraries*. PhD thesis, 2001.

[12] Anthony Tomasic and Hector Garcia-Molina. Query processing and inverted indices in shared nothing text document information retrieval systems. *The VLDB Journal*, 2(3):243–276, 1993.

[13] Wensi Xi, Ohm Sornil, Ming Luo, and Edward A. Fox. Hybrid partition inverted files: Experimental validation. In *ECDL '02: Proceedings of the 6th European Conference on Research and Advanced Technology for Digital Libraries*, pages 422–431, London, UK, 2002. Springer-Verlag.