Inf Retrieval (2007) 10:205–231
DOI 10.1007/s10791-006-9014-4

A pipelined architecture for distributed text query evaluation

Alistair Moffat · William Webber · Justin Zobel · Ricardo Baeza-Yates

Received: 28 September 2005 / Accepted: 25 August 2006 /
Published online: 5 October 2006

Abstract Two principal query-evaluation methodologies have been described for cluster-based implementation of distributed information retrieval systems: document partitioning and term partitioning. In a document-partitioned system, each of the processors hosts a subset of the documents in the collection, and executes every query against its local sub-collection. In a term-partitioned system, each of the processors hosts a subset of the inverted lists that make up the index of the collection, and serves them to a central machine as they are required for query evaluation.

Alistair Moffat was supported by the Australian Research Council, the ARC Special Research Center for Perceptive and Intelligent Machines in Complex Environments, and the NICTA Victoria Laboratory.

William Webber and Justin Zobel were supported by the Australian Research Council.

Ricardo Baeza-Yates was supported by Grant P01-029-F from Millennium Initiative of Mideplan, Chile; and by the University of Melbourne as a visiting scholar at the time this project was undertaken.

A. Moffat (✉) · W. Webber
Department of Computer Science and Software Engineering, The University of Melbourne, Melbourne, Australia 3010
e-mail: alistair@csse.unimelb.edu.au

W. Webber
e-mail: wew@csse.unimelb.edu.au

W. Webber · J. Zobel
School of Computer Science and Information Technology, RMIT University, Melbourne, Australia 3001

J. Zobel
e-mail: jz@cs.rmit.edu.au

R. Baeza-Yates
Center for Web Research, Department of Computer Science, University of Chile, Santiago, Chile

Present address:
R. Baeza-Yates
Yahoo! Research, Barcelona, Spain
e-mail: ricardo.baeza@upf.edu

In this paper we introduce a pipelined query-evaluation methodology, based on a term-partitioned index, in which partially evaluated queries are passed amongst the set of processors that host the query terms. This arrangement retains the disk read benefits of term partitioning, but more effectively shares the computational load. We compare the three methodologies experimentally, and show that term distribution is inefficient and scales poorly. The new pipelined approach offers efficient memory utilization and efficient use of disk accesses, but suffers from problems with load balancing between nodes. Until these problems are resolved, document partitioning remains the preferred method.

Keywords Distributed retrieval · Text searching · Index representations

1 Introduction

Ranked query evaluation in text databases is a straightforward application of inverted indexes. The inverted list for each query term is fetched, and the per-document weights in the list are used to update per-document values in a set of accumulators. The accumulators with the largest values are then identified (possibly after a normalization step), and the corresponding documents are fetched and presented to the user (Baeza-Yates and Ribeiro-Neto, 1999; Witten et al., 1999; Zobel and Moffat, 2006).

Several strategies can be used to improve the efficiency of query evaluation. The main intent of many of these strategies is to reduce the volume of index information fetched and processed. Options include compression, to make better use of bandwidth; list reordering, so that high weight postings are immediately available at the front of lists; and various approximations, to allow the computation itself to be streamlined. Such techniques can involve complex trade-offs. For example, with some strategies the similarities computed during query evaluation determine in an on-the-fly manner how much additional index information needs to be retrieved, meaning that the cost of query evaluation can vary considerably from one query to the next (Persin et al., 1996; Hawking, 1998; Anh et al., 2001; Anh and Moffat, 2006).

As another example of the complexities involved in some of the design decisions for text retrieval, consider how memory might be used. A simple choice is to store the entire collection vocabulary in memory, on the basis that even a typical desktop PC can comfortably hold ten million terms in memory along with the associated housekeeping information, thus saving a disk access for each term during query processing. However, in a stream of queries some terms are more common than others, meaning that much of the vocabulary may be accessed rarely, if at all, and the hundreds of megabytes required for the vocabulary can be deployed for other facets of query processing. In particular, if the system allows query parallelism, then the same memory can be used for the sets of accumulators required by each query process; or for caching of inverted lists; or for caching answers to common queries; or for caching documents themselves. Different trade-offs between kinds of caching might significantly affect performance, with the appropriateness of the decisions depending on factors such as the query arrival rate. But disk fetch time is likely to remain a key bottleneck in all large retrieval systems.

The capacity of a single-server text database system is limited in two quite different ways. If query throughput rates are a problem, then the system can be *replicated*, and a *receptionist* process used to direct queries to one of a set of mirrored systems, each of which is completely capable of answering any query. Replication involves relatively small overheads, and approximately linear gains in throughput capacity can be expected as a function of the number of processors used. On the other hand, if data volume is a problem, then parallelism must be used in a rather different way, via *distribution*. In a distributed system, the receptionist passes each query to some or all of the processors, and synthesizes a system-wide answer from the partial answers computed from each part of the system. Because of the need for additional processing at the receptionist, linearity of performance cannot be assumed. Moffat and Zobel (2004) consider in more detail the distinction between replication and distribution in text database systems. For the purposes of this article it is sufficient to note that distribution—required primarily when data volumes beyond that which can be handled on a single machine must be accommodated—is the more interesting problem from an algorithm design point of view.

Two standard techniques for index organization in distributed environments have evolved: *document partitioning*, in which the document set is split over the set of processors, and each processor stores a full index for its subset of the documents; and *term partitioning*, in which the index is split over the set of processors, and each processor stores full index information for a subset of the terms. Each of these two organizations has a corresponding query processing regime, described in more detail below. Once storage-related issues have been resolved by distribution, query throughput rates can then be resolved by replicating the entire system. This is the approach used by, for example, the `Google` web search service (Barroso et al., 2003).

In this paper we introduce a novel *pipelined* approach for evaluating queries against a term-partitioned index, in which "bundles" that represent partly evaluated queries are shipped between processors. Pipelining retains the attractive disk read and memory characteristics of term partitioning. Compared to document partitioning, the anticipated advantages of the new approach include fewer disk accesses, as lists are not stored in fragments; and more efficient use of main memory, as each machine stores a smaller vocabulary. The fundamental question that arises is then: do these hypothesized advantages convert into improved query throughput rates?

To evaluate different distribution approaches, we have carried out comprehensive experiments using a large document collection, a realistic query log, and carefully tuned implementations. The experiments show that the new pipelined system is more efficient and scalable than term partitioning. However, poor workload distribution means that pipelining does not scale as well as document partitioning, and unless a better load balancing system can be devised, document partitioning remains the distributed architecture of choice when measured by peak query throughput. On the other hand, the experiments confirm our claims for disk access loads and memory utilization, showing that in environments where query term frequency is not so skewed—which might be possible, for example, if answers to frequent queries are cached—pipelining may well be the method of choice.

The focus in this paper is on the use of cooperating, tightly-coupled computers on a high-speed network, in applications with large amounts of indexed data, such as web search engines. Other literature explores issues such as retrieval in the context of heterogeneous distributed databases, potentially held at remote locations, and managed independently; such environments are beyond the scope of this paper.

Our experiments are not the first comparison of document partitioning and term partitioning. Ribeiro-Neto and Barbosa (1998) and Badue et al. (2001), for example, found term partitioning to be superior, while Tomasic and García-Molina (1993) found document partitioning to be best. The results of Jeong and Omiecinski (1995) point both ways. Xi et al. (2002) investigate similar issues, considering multi-threading and using 100 GB of TREC data; the results (with what appears to be a low computational workload) favor their alternative partitioning scheme (see also Sornil (2001)). These contradictory findings underline the need for careful experimental design.

The remainder of the paper is organized as follows. In Section 2 we describe the two standard techniques for partitioning large collections so as to allow distributed retrieval. The new technique is introduced in Section 3. In Sections 4 and 5 we present an experimental comparison of throughput rates, and the conclusions that we draw from the observed behavior of the methods. Finally, in Section 6, we analyze the workload balancing and disk read characteristics of document partitioning and the new pipelining technique.

2 Distribution techniques

The index of a large document collection can be distributed across cooperating processors in two distinct organizations—via a partition based on the documents of the collection, or a partition based on its vocabulary. The next two subsections describe these organizations, and the matching query processing regimes.

2.1 Document-partitioned indexing and querying

A straightforward way of distributing the retrieval task is to allocate each computer, or *server*, a defined fraction of the documents and then build an index for each local document set (Harman et al., 1991; de Kretser et al., 1998; Cahoon et al., 2000). Each index consists of a complete vocabulary for the documents on that computer and, for each term in the vocabulary, an inverted list recording the documents containing the term and (if phrase querying is to be supported) the positions in each document at which the term occurs. Queries are accepted by a *receptionist* process, which broadcasts them to each of the k servers in the cluster. Assuming that the top r documents are to be returned to the user, each server computes a list of $r' \geq r/k$ highest-rank documents. The set of lists from the servers are then merged by the receptionist to yield a single ranked list of length $k \cdot r'$ from which the top r are proposed as answers.

One of the computers in the cluster might act as the receptionist, or it might be a separate computer external to the cluster. The receptionist task might also be distributed across all of the servers, and run as a low-impact process on each of them. In the latter case, all that is required is that the individual ranked lists corresponding to each query be returned to the receptionist process that initiated that query.

In this architecture, some of the costs of retrieval are duplicated. For k servers and a query of q query terms, $k \cdot q$ inverted lists must be fetched, and, even though I, the total volume of data handled, may be little different to the centralized case, there are potentially k times as many disk seeks. The additional seeks are in parallel and do not slow the elapsed time to resolve the query, but they do mean that execution of a query stream on k computers is unlikely to be k times faster than execution of the same query stream on one computer, unless there are significant caching benefits accruing from the more efficient memory usage. In addition, some of the computation done by the computers is unnecessary, as only a subset of the $k \cdot r'$ documents ranked by the k servers is retained by the receptionist and presented to the user in the final list of r highly ranked documents, and $r' = r$ is a typical implementation assumption. Table 1, taken from Moffat and Zobel (2004), summarizes the main costs associated with querying using document-partitioned distribution, and compares them with the costs associated with a monolithic system executing on a single processor.

Document partitioning has several practical strengths. It can be used in a loosely-coupled environment, where the databases are independently managed, and perhaps even employ different software and ranking heuristics. Insertion of new documents in a document-partitioned collection is straightforward, as all the information about a given document is held on one

Table 1 Comparison of costs associated with monolithic, document-partitioned, and term-partitioned retrieval, when a query of q terms is processed with a k-way data partition, to determine a ranked list of r answers. Quantity I is the sum of the lengths of the inverted lists for the query terms, counted in pointers. (Adapted from Moffat and Zobel (2004))

Performance indicator	Monolithic system	Document partitioned	Term partitioned
Number of servers active on query	1	k	q
Per processor			
Disk seeks and transfers	q	q	1
Index volume transferred from disk	I	I/k	I/q
Number of documents scored	r	r	0
Plus			
Network volume	n/a	kr	I
Computation load at receptionist	n/a	kr	$I+r$
Total cost	$I+q+r$	$I+kq+kr$	$I+q+r$

server. Query evaluation can proceed even if one of the servers is unavailable, in a graceful rather than abrupt degradation of service. All querying modes, including phrase queries and Boolean retrieval, are easy to support. The same cluster can be used to produce surrogate document representations for the presentation of results as was used for the query evaluation. Finally, the simple strategy of copying documents from one server to another provides useful redundancy (Clarke et al., 2003).

Note that, in some of the relevant literature, document partitioning is referred to as the *local index* approach.

2.2 Term-partitioned indexing and querying

An alternative to document partitioning is to use a term-partitioned index (Jeong and Omiecinski, 1995; Ribeiro-Neto et al., 1999; Badue et al., 2001). In this approach, each server is responsible for maintaining all index information pertaining to a given subset of the terms. To evaluate a query, the receptionist asks the appropriate servers to supply the terms' inverted lists. The receptionist then combines the inverted lists to generate the ranking. Term partitioning is referred to as the *global index* approach in some of the relevant literature.

Compared to document-partitioned query evaluation, disk activity—which has the potential in large systems to be a significant query-time cost—is greatly reduced. In the term-partitioned approach, the servers are little more than disk controllers, and most of the computation is undertaken by the receptionist. Compared to a monolithic system, the cost of disk accesses is replaced by network transfers, possibly preceded by remote disk accesses.

The load imbalance is a significant disadvantage to term distribution, and there is a real risk that the receptionist process will become a bottleneck, and starve the servers of useful work. There are also other disadvantages. One is that the individual servers do not have enough information to reliably determine how much of each inverted list must be retrieved. The receptionist may be able to estimate this information, but only if it maintains a complete vocabulary with detailed statistics. A related disadvantage is that evaluation of phrase or Boolean queries is inefficient, as it involves transmission of complete inverted lists including word positions.

Another disadvantage is that index creation is more complex. For the experiments described below we built a monolithic index and then partitioned it; more generally, each server needs to separately index a disjoint part of the collection, and then negotiate and execute a set of pairwise exchanges of list fragments (Ribeiro-Neto et al., 1999). Either way, an extra processing stage is required compared to a document-partitioned index.

Results from an implementation described by Badue et al. (2001) suggest that a term-partitioned system is able to handle greater query throughput than a document-partitioned one, assuming the receptionist has a large memory and that the distribution of query terms is skewed, allowing effective caching. However, it is not clear that term-partitioned distribution is more efficient than having a single computer with multiple disks and disk controllers. Nor is it clear how well this strategy would perform for large collections, and the issue of scalability was not addressed in the experiments carried out by Badue et al. The third column of Table 1 shows the per-query costs associated with this evaluation mode in a term-partitioned index. At face value, term partitioning appears to have an advantage over document partitioning, because the latter includes a factor of k in the total execution cost. However the difference is illusory, since there is a close relationship between k and I that is not explicitly noted in the table entries.

2.3 Other approaches

An alternative to term or document partitioning is described by Xi et al. (2002) (see also Sornil (2001)), in which unit-length inverted list fragments are distributed across all servers, and query evaluation involves fetching all pieces of all lists corresponding to query terms. Given the problems described later with term partitioning, and the observation that this method has the additional disadvantage of more disk accesses and no clear advantages, we have not investigated it further.

2.4 Threading

An issue that arises in all methods, but is particularly important to term partitioning, is the impact of multi-threading of the execution flow. If execution is single-threaded, then only one query is active in the receptionist at any given time. For a query of q terms and a cluster of $k > q$ servers, single-threading must, of necessity, imply that in a term-partitioned system at least $k - q$ servers are idle while this query is being handled. Moreover, q is likely to grow slowly (if at all) as the collection grows, whereas k grows linearly. Hence, in a single-threaded term-partitioned system, an increasing fraction of the servers are idle at any given moment in time, and overall efficiency suffers. If server starvation is not to take place, this reasoning suggests that the receptionist must operate at least k/q_{avg} threads or that there must be multiple receptionists, where q_{avg} is the average number of terms per query.

In a document-partitioned system the difference between multi-threading and single-threading is also important if the primary performance criterion is throughput rather than average query response time (Orlando et al., 2001). Our experiments, discussed later, demonstrate the importance of multi-threading in both distributed and monolithic retrieval systems.

2.5 Calculating similarity scores

Another issue that arises in all methods is the process of matching queries to documents. In a typical search engine the indexes are used in a variety of ways. Most common are ranking queries, in which (notionally) every document is assigned a similarity score with regard to the query and the top-scoring documents are returned to the user. Some ranking queries involve phrases, in which a Boolean process is used to intersect the inverted lists of the terms in the phrase, taking note of word positions to determine which documents contain the word as a phrase. We do not consider evaluation of phrase queries in this paper.

The most effective measures for computing the similarity of a query and document combine several forms of evidence of relevance, including how often each query term t occurs in the document d (denoted $f_{d,t}$), how rare the terms are in the collection overall (a reciprocal function of F_t, the frequency of t in the collection), and how long the document is. In the experiments in this paper, we calculate similarity using a Dirichlet-smoothed language model (Zhai and Lafferty, 2004).

A straightforward and efficient way to use inverted lists to determine a ranking is to process each inverted list in turn, starting with the list for the rarest term and proceeding to the most common term. Each document referenced in an inverted list is allocated an accumulator variable; and query evaluation proceeds by using the statistics held about each document in each inverted list to add to each accumulator a contribution corresponding to the term's importance in that document. The maximum number of accumulators allocated may be limited by one of a number of strategies, such as allowing only rare terms to create new accumulators, and restricting common terms to updating existing ones (Moffat and Zobel, 1996), or by dynamically setting a threshold for accumulator creation and retention (Lester et al., 2005a). The accumulators are maintained in a data structure, such as an ordered linked list, that allows each additional list to be merged against it.

Other inverted list structures, such as impact ordering (Anh et al., 2001), allow faster query evaluation. However, the standard structures described above are the norm in current search engines, and have practical advantages such the ability to handle Boolean and phrase queries, and simplicity of update.

3 Pipelined distribution

Our hypothesis is that in the document-partitioned approach there is a performance degradation caused by duplicated disk transfers. Those transfers are avoided in the term-partitioned arrangement, but a competing risk is introduced, namely that the receptionist becomes overworked, and the servers are starved and unable to make an effective contribution. This section explores an alternative approach that is designed to avoid both of these problems.

3.1 Query bundles

In the new *pipelined* approach we again make use of a term-partitioned index, but require that the servers take more responsibility for query resolution, to relieve the load on the receptionist. The receptionist maintains a global vocabulary that indicates which server is responsible for each term, and on receipt of a new query, creates a *query bundle* that includes a list of query terms, a processor routing sequence that indicates which hosts hold those terms, and an initially-empty accumulator data structure.

Query evaluation proceeds by passing the partially processed query bundle through a sequence of servers. Each server that holds information about the terms in a particular query receives the query bundle, folds in its contribution to the ranking based on its fragment of the global index, and then passes the query bundle on to the next server in the routing sequence. The last server in the sequence extracts the r top-ranked document numbers and returns them to the receptionist, which hands them on to the part of the system responsible for presenting answer lists. The similarity scores so obtained are close to being identical to those that would be calculated by a monolithic system, with the sole difference being that terms are applied on a per server basis, meaning that when the query bundle arrives at a server, all terms available at that server are applied to it. As a result, queries with multiple terms present on the same server may not have terms applied in strictly increasing F_t order, which might affect the final similarity scores in a limited-accumulators environment.

Figure 1 provides details of the pipelined approach to distributed query evaluation, and Fig. 2 shows how a query of three terms might be routed through a cluster of five servers.

1. The receptionist receives the query, parses it, applies any stop-word heuristics, and consults the vocabulary to identify the collection frequency of each term, and the identity of the server that stores the index information for that term.
2. The receptionist assembles a query bundle, which contains a list of the terms in the query; a routing list of nodes storing those terms, in decreasing order of lowest collection frequency order; and an accumulator data structure, initially empty.
3. The receptionist sends the query bundle to the first server on the routing list.
4. Each server that receives the query bundle then
   (a) Fetches the inverted lists corresponding to any query terms for which it holds index information.
   (b) Updates the set of accumulators by computing partial similarities using those inverted lists, using dynamic thresholding to limit the number of accumulators.
   (c) Updates the query bundle, and, if query terms remain, transmits it to the next server on the routing list. Otherwise, it returns to the receptionist the document identifiers corresponding to the top r accumulators.
5. The receptionist prepares an answer list for the r top-ranking documents.

Fig. 1 Pipelined query evaluation in a distributed computing environment

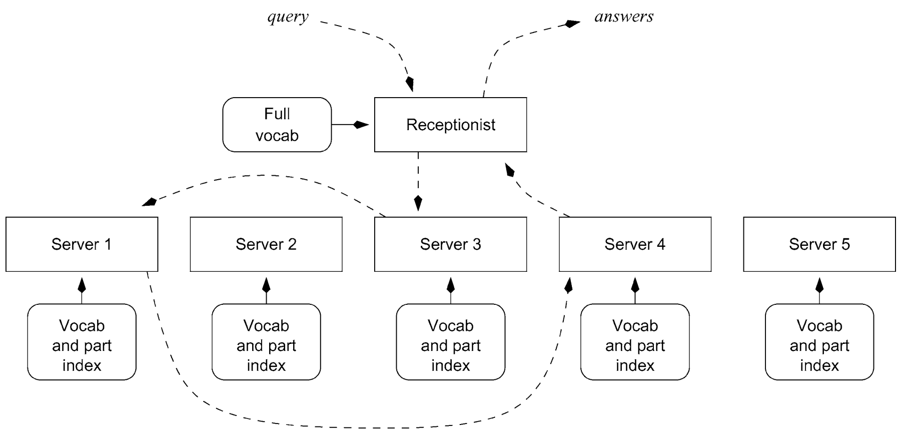


Fig. 2 Pipelined query processing with a term-partitioned index. In this example, the query contains terms that necessitate routing the query bundle through processors 3, 1, and 4 in a system containing five servers and five index partitions

3.2 Relative merits

In terms of Table 1, pipelining requires that at most q processors act on each query; that, when the query terms are on different servers, each processor that works on the query carries out one disk seek to fetch an average of I/q index pointers; and that the total network volume is given by $(q-1)\cdot A$, where A is the cost of moving the accumulator data structure. The total cost is thus given by $I+q+r+(q-1)\cdot A$, with the difference between it and the other methods determined by empirical constants, and the degree to which parts of the computation can be parallelized.

A potential disadvantage of pipelining is that each transfer of a partially evaluated query might require significant network resources—at the very least, a relatively large set of numeric accumulator values must be communicated, accounted for as A in the analysis of the previous paragraph. On the other hand, with a suitable dynamic query pruning regime, this set could easily be smaller than an inverted list for a common term. Moreover, just two or three such transfers are required during execution of a typical query, whereas many messages are required in a document-partitioned system. In the pipelined implementation, this cost is minimized by difference-encoding document numbers, and quantizing scores to 256 distinct values, thereby shrinking twelve bytes of in-memory data per accumulator to around three bytes when serialized for transfer.

A drawback of the pipelined approach is that it does not provide ready support for phrase or proximity queries, which are most conveniently handled by direct merging of all applicable index lists, rather than by processing the lists in turn—as in the case of similarity queries—and aggregating numeric scores.

Another potential problem with pipelined evaluation is that the processing load can become unevenly distributed between nodes. In an extreme case, a stream of queries might consist of nothing but a single repeated term. In a pipelined system, only one node would be active; in a document-partitioned one, the processing load would still be evenly distributed between nodes (assuming that the term is evenly frequent throughout the partitions of the collection). Even in a more typical query set, term frequencies are generally highly skewed, both in the query set itself and in the collection, and an even balancing of work between nodes cannot be assumed.

Pipelined evaluation has several potential advantages. There is the possibility of early termination of query evaluation in step 4c, as in standard inverted indexes. The receptionist does much less work, and if necessary, query evaluation can be terminated if the system becomes congested, saving processing of some inverted lists. Finally, with an accumulator limit enforced, and compression applied to the relevant data structures, the cost of shipping the query bundles may be significantly less than that of shipping the corresponding inverted lists.

4 Experimental framework

Critical to any experiment is a clear understanding of what is being measured; of what the variable factors are and how they are controlled; and of what factors that could be variable are in fact being held constant (Moffat and Zobel, 2004). This section considers previous experimental investigations of distribution, and then describes the experimental environment used. Section 5 below then presents results for monolithic, document-partitioned, term-partitioned, and pipelined query evaluation.

4.1 Previous investigations

Previous investigations of distributed retrieval have typically focused on the ratio of speedup that is possible when additional servers are applied to a fixed task, such as querying a certain volume of text; see for example Cahoon et al. (2000), MacFarlane et al. (2000), Badue et al. (2001) and, Cacheda et al. (2004). But there are drawbacks to this approach. If the time taken to process a fixed set of queries against a fixed collection is measured as $k = 1, 2, 3, \ldots$ computers are used, then there are two variables—additional processors are available, and additional disk and memory resources are being deployed to handle the same volume of data. It is thus hard to establish the extent to which any measured speedup is a consequence of genuine processing parallelism rather than a consequence of fewer disk seeks because of increased memory for caching.

To avoid this ambiguity, the results below make use of document collections of a range of sizes, and the only inputs that we hold constant are the average number of distinct terms in each of the queries in the input stream, and the number r of highly-ranked documents that are returned as the answers to each query. That is, we suppose that queries are the same length, regardless of the size of the collection that is being queried, an assumption that has some basis in observed user behavior (Spink et al., 2001); and that $r = 1{,}000$ answers are required for each query.

Web-based applications typically require far fewer than $r = 1{,}000$ answers per query. However, computation time is only a small factor in the overall processing cost, and identifying 1,000 documents (rather than 20, say) does not disadvantage any of the distribution methods more than the others.

4.2 Measurement

There are several quantities that can be measured and reported in an investigation of distributed text retrieval. For example, we might be interested in knowing how many seconds it takes, on average, for queries to be answered, assuming that each query arrives when the system is idle. Alternatively, we might be interested in knowing what the peak query processing rate is, to get an idea of the maximum throughput possible with the available resources. In this paper, we concentrate on this latter measure, the maximal throughput of the system.

One important concern with multi-dimensional throughput experiments—in which the number of processors might be varied independently of the volume of data being handled—is to be sure to compare like with like. To allow appropriate comparisons, the numbers reported in the tables below all have the dimension "(queries × terabytes)/(machines × elapsed seconds)". That is, if an arrangement of k processors is able to handle q queries in s seconds against a collection of T terabytes, then the number $(q \times T)/(k \times s)$ is computed as a rate at which useful work is being done, and referred to as the *normalized throughput rate*. Roughly speaking, for a fixed query length, the total work to be performed grows as a linear function of the number of queries and of the size of the collection, the latter because the lengths of the inverted lists required for typical query terms contain a constant fraction of the documents in the collection, regardless of collection size. Dividing this workload by the number of processors used, and the time taken to do it, gives a "bang per buck" ratio that can be used to assess the relative efficiency of competing approaches. Larger rates mean higher throughput, and hence better performance.

Use of this metric allows a wide range of comparisons:

- With the number of processors held constant, growth in the size of the document collection allows any economies of scale caused by collection size to be quantified;
- With the collection size held constant, growth in the number of processors used allows overheads due to inter-process communication to be measured; and
- With collection size and number of processors varied in tandem, the extent to which any technique is scalable at a constant efficiency ratio can be established.

In factoring the number of machines in to this calculation, we are implicitly assuming that no finer unit of resource is available, and that all machines are equivalent. This is clearly not the case, and a more precise model might, for example, account for processors (or even total gigahertz) and disk storage independently. Ultimately, the fundamental unit of cost is dollars, and if we wish to be meticulous in the accounting, an allowance should be made for software and air conditioning and insurance and so on, a rather complex analysis. Instead, we stop at "machines", and assume that there is some unit cost associated with each unit of combined disk storage and processing power at a given clock speed.

4.3 Hardware and software

The hardware used for all of the experiments reported in this paper is a Beowulf-style cluster of 8 computers, each a 2.8 GHz Intel Pentium IV with 1 GB of RAM and 250 GB local SATA disk, connected by a 1 Gbit local network. These nodes are collectively served by a dual 2.8 GHz Intel Xeon with 2 GB RAM running Debian GNU/Linux (sarge), with a 73 GB SCSI disk for system files and twelve 146 GB SCSI disks for data in a RAID-5 configuration. In each case, the receptionist process was executed on the interface machine, with as many as eight of the cluster machines used as servers. The retrieval system makes use of a document-level inverted index, with a byte-aligned code used for index compression (Scholer et al., 2002), and is derived from the Zettair system, see `http://www.seg.rmit.edu.au`. Index lists are stored in document number order, and represented as a sequence of alternating d-gaps and corresponding $f_{d,t}$ values. Word positional offsets are not stored.

Queries were evaluated using a dynamic thresholding mechanism to maintain the number of document accumulators at or near an accumulator target (Lester et al., 2005a). Under this mechanism, query terms are processed in increasing order of term frequency F_t, and, up to the accumulator target, every term occurrence creates a document accumulator. Once the accumulator target has been reached, a threshold is set on the similarity contribution for new accumulators to be added, and for existing ones to be retained. This threshold is then dynamically adjusted to keep total accumulator numbers close to the target amount as the remaining query terms are processed. For efficiency reasons, the pipeline model did not strictly follow F_t ordering of all terms; instead, nodes were visited in increasing order of the lowest F_t query term held on each node, with remaining terms on the same node being processed before shipping the accumulator bundle to the next node. The difference in resultant document rankings was slight, and is quantified later in the paper. The deemed result of processing a query was a list identifying the top-ranked $r = 1{,}000$ documents for that query, based on the accumulator values. Effectiveness results are given in a later section.

All experiments were carried out five times, and the throughput rates reported represent the average of the rates attained over the five runs. In preliminary experiments, we observed that experimental times varied by as much as 15%, depending on where physically the blocks of the inverted index were placed on disk (Webber and Moffat, 2005). To avoid such

Table 2 The various sample document collections. The final row shows the average volume of compressed index data processed when executing each of the 10,000 test queries on that fractional collection

	Collection						
Attribute	TB/64	TB/32	TB/16	TB/08	TB/04	TB/02	TB/01
Size (GB)	6.7	13.3	26.6	53.2	106.4	212.9	425.8
Documents ($\times 10^6$)	0.39	0.79	1.57	3.16	6.31	12.60	25.21
Terms ($\times 10^6$)	4.89	8.15	13.34	21.68	35.13	56.19	87.74
Index (GB)	0.3	0.7	1.2	2.4	4.5	8.7	16.6
Query cost (MB)	0.03	0.06	0.13	0.25	0.51	1.01	2.02

problems, care was taken throughout the experimentation to make sure that all index files were served from similar physical locations on disk, and with minimal fragmentation.

4.4 Test data

The various document collections used in our experiments are all derived from the TREC Terabyte (or GOV2) collection built in 2004 by crawling a large number of sites in the .gov domain (see http://trec.nist.gov). In total, the collection contains 426 GB. The full collection is referred to in this paper as TB/01. To form sub-collections of different sizes, a round-robin approach was adopted, and TB/01 split into approximately evenly sized parts by selecting every *n*th file, of the 27,000 files that make up the complete collection, for different values of *n*. For example, a "half" collection containing approximately 213 GB was formed by taking every second file, and is referred to as TB/02. Similarly, the TB/08 division refers to a one-eighth collection, containing approximately 54 GB of data. Table 2 summarizes the seven collections used, and lists some pertinent statistics for them. Note the linear growth in query cost as the collection expands (the quantity denoted I in Table 1), and that the number of distinct terms in the collection also grows, but sub-linearly, as the collection grows.

Each collection was document-partitioned using a similar procedure to that used for creating the smaller collections from the full TB/01 collection. To create an eight-way partitioning of a given collection, for instance, every eighth file of the collection was extracted into the first partition, every eighth plus one into the second, and so on. We note in passing that this method of partitioning the collection provides for a more homogeneous spread of term frequencies between the sub-collections, and therefore for a more even balancing of workload, than simpler methods might, such as dividing the collection by time of document creation or discovery.

Term partitioning was achieved by calculating a hash value for each term, and assigning the term to a partition by taking the modulus of this hash value over the number of partitions. No attempt was made to balance partitions by reference to the frequency of terms, either in the collection or in a training query set. Exactly the same partitions were used in the term-partitioned and pipelined runs. Section 6 discusses some of the consequences of this approach.

4.5 Test queries

A perennial problem in information retrieval research is finding suitable query sets. The GOV2 collection consists solely of web pages and documents crawled from US government web sites, and the query logs that are publicly available, such as the Excite or AltaVista logs, are inappropriate, consisting for the most part of queries unlikely to be posed to a US government site. In particular, queries taken from a whole-of-web search service are likely to contain terms that are relatively rare in GOV2, and as a result are likely to return fewer relevant documents.

To address this issue, we generated a synthetic query stream, based upon a real-world query log and a matching general web document collection (Webber and Moffat, 2005). The query log was the Excite97 log, consisting of around 1 million queries submitted to the Excite search engine on a single day in 1997. The collection was the wt10g collection (Bailey et al., 2003), crawled from the general web in 1997 and thus a reasonable match for the Excite97 queries. The Excite log was sorted in arrival order, with phrase and boolean queries converted to non-conjunctive queries. It was then stopped with a list of 300 stop-words, to a minimum of three words per query, with “the”, “and”, “a”, “are”, “of”, and “is” unconditionally stopped. Terms longer than fifteen characters, including URLs, were removed. Each query term in the cleaned log was then assigned a translation term in the vocabulary of GOV2 of approximately the same relative collection frequency as the original term had in wt10g. Translation terms were picked to maximize the number of documents in which all terms for each query co-occurred. The resulting synthetic query stream preserves identical query length and query term frequency distributions as the original, and has similar query term f_t and co-occurrence distributions (with respect to wt10g). The drawback of this approach is that the resulting queries do not “make sense” (for example, the common query term “sex” was translated into “vhs”); however, from the point of view of efficiency experiments, they preserve the essential features of a real-world query log.

The first 20,000 queries from the resulting synthetic log were run against each of the systems to measure their performance. The elapsed time between the release of the 10,001st query into the system and the receipt of the result for the final completed query (not necessarily the 20,000th query, due to the parallelized nature of query processing) was measured, and used to calculate a normalized throughput rate. By delaying the timing until the second 10,000 queries, the system is able to make full use of main memory to cache index information. Additionally, before starting each run, the index files were mapped into the system file cache on each node, up to the limit of the node’s main memory. This is particularly important where the index on a node was smaller than the node’s main memory: the entire index could fit into memory, but since new terms keep occurring in queries, the 10,000-query warmup run would be unable to actually load all of the index into memory.

The 10,000 timed queries contained an average of $q_{avg} = 2.15$ terms per query. The last row of Table 2 shows the index volume required to answer each query, measured as the average number of megabytes of compressed index lists associated with the terms of each query. Only the index processing stages of each query evaluation were performed, through until the moment when a list of $r = 1{,}000$ documents had been identified as answers. Note also that the indexes were completely unstopped, and that no stemming was performed on either index or queries.

4.6 Accumulator limits and retrieval effectiveness

One of the key execution-time variables in the experimental software is the number of accumulators permitted to the similarity computation (Lester et al., 2005b). On a monolithic system, the main benefit in limiting the number of accumulators is in restricting the amount of memory used during the query evaluation. That benefit carries through to each of the distribution methods examined here. In addition, the pipelined method transfers in-progress accumulator bundles between nodes, and is additionally sensitive to changes in the accumulator target. Setting the accumulator target too low can adversely affect retrieval effectiveness, by excluding documents whose low similarity on high-discriminating terms is compensated by high similarity on low-discriminating terms; setting it too high potentially wastes resources. The ideal accumulator target is as low as is consistent with acceptable retrieval effectiveness compared to an unrestricted evaluation.

In the presence of an accumulator target or limit, different distribution methods can produce slightly different retrieval results. For document partitioning, each node locally carries out an independent query evaluation with its own accumulator target. In these experiments, we assigned each document-partitioned node a proportion of the total accumulators: if there were k nodes, and A accumulators system-wide, then each node would get A/k accumulators. For the pipelined system, strict in-F_t-order processing is not followed, as described earlier. In contrast, term partitioning replicates precisely the results of a monolithic system with the same number of accumulators, regardless of the number of nodes.

In their study of accumulator restriction mechanisms in a monolithic environment, Lester et al. (2005a) found that the GOV2 collection requires approximately 400,000 accumulators if full retrieval effectiveness is to be achieved, when assessed using the 2004 TREC Terabyte topics and mean average precision; but also found that the loss in effectiveness in reducing this to 100,000 was insignificant. We thus took as a starting point the use of 100,000 accumulators, and verified, using the same queries, that consistent retrieval effectiveness was obtained using the three distributed paradigms.

We also measured the effect of the target of 100,000 accumulators on the synthetic query stream. In the absence of relevance judgments, we estimated the discrepancy between full evaluation and accumulator-limited evaluation using the notion of ranking *dissimilarity*, and compared different methods and targets against the baseline run of a monolithic system with an unlimited number of accumulators.

Dissimilarity is established by comparing a ranking against a reference one. Items that are at the same ranked position in both lists contribute zero to the score, whereas items that have changed rank—or disappeared entirely—contribute a positive amount that inversely depends on their rank position. This metric assigns greater "difference" to variations near the top of the ranking than to variations at the bottom; two rankings that are completely disjoint are assigned a score of 1.0, while two rankings that are identical are assigned a score of 0.0. The exact calculation used is dependent on a damping factor T; if the item in position p of one ranked list is in position $\ell(p)$ in the other, then it contributes

$$\left|\frac{1}{T+p} - \frac{1}{T+\ell(p)}\right|$$

to the dissimilarity score. These contributions are summed over all of the r items in the ranked list, and then normalized by the maximum score that could be attained if all r items did not appear, to obtain a normalized dissimilarity.

The accumulator limit of 100,000 was again confirmed as being a suitably conservative value, and only a small variation in rankings resulted.

Having selected 100,000 as the number of accumulators for the full TB/01 collection, it was then necessary to consider whether to reduce the limit for the smaller collections, and if so, by how much. It seems reasonable to vary the number of accumulators in proportion to the collection size; as we halve the collection, we might halve the number of accumulators. This is also attractive as it preserves an important relationship within the document-partitioned experiments, which is that each node in a k-node partitioning of a collection does as much work as each node in a $(2k)$-node partition of twice as much initial data. The effect of scaling the number of accumulators with collection size on normalized dissimilarities is shown in Table 3. Scaling the accumulators gradually increases dissimilarity, a consequence of the fact

Table 3 Normalized dissimilarity over 10,000 synthetic queries, first holding accumulators constant at 100,000, and then scaling accumulators with collection size, in all cases using a monolithic system. Dissimilarity values are relative to a baseline of unlimited accumulators. For example, `TB/04` is processed with 100,000 and then 25,000 accumulators. A score of zero indicates that the ranking is indistinguishable from that obtained with unlimited accumulators; a score of 1.0 indicates that the two rankings in question are disjoint

		Accumulators	
Fraction f	Collection	100,000	100,000/f
1	`TB/01`	0.022	0.022
2	`TB/02`	0.011	0.026
4	`TB/04`	0.004	0.031
8	`TB/08`	0.001	0.036
16	`TB/16`	0.000	0.045
32	`TB/32`	0.000	0.055
64	`TB/64`	0.000	0.057

that the number of results is not scaled, but kept constant at 1,000. However the overall level of dissimilarity remains acceptable for our purposes and, in terms of execution time, errs in favor of the reference document-distributed scheme. The scaled approach to accumulator limiting was thus used in the experiments described below.

4.7 The effect of threading

The need for a threaded implementation of the receptionist and server software was raised earlier. Threading is a valuable mechanism for exploiting potential parallelism, even in a monolithic system, as it allows evaluation of one query to proceed while another is blocked pending a response to a disk or network request. The next set of experiments was designed to investigate the extent to which query concurrency was necessary for high throughput rates, and to choose the number of threads to use in the subsequent experiments.

Running with the full `TB/01` collection, and $k = 8$ nodes, we measured the throughput of each distribution method with different values of t, the number of queries concurrently executing system-wide. In the pipelined system, a query can only be in process on a single server at any given point in time, so the average number of threads per server is t/k. In the document-partitioned system, each query is simultaneously executed on every server, so the number of threads on each server is potentially t. In practice, because some threads finish their work before others, the actual active load per server was around 75% of this figure.

Pipelining's throughput rises consistently as t is increased, whereas document distributed throughput peaks at $t = 32$. The difference is a consequence of the average load per server, and demonstrates the inhibiting effect of system load. Term partitioning plateaus after $t = 4$ threads, at which point the single evaluator node becomes saturated. Note that, with a single thread, document partitioning achieves around 40% of maximum throughput, whereas pipelining achieves less than 15%. The difference between these two approaches underlines the importance of allowing parallel execution in the pipelined system.

While Table 4 suggests that using 64 threads for the pipelined system might give a marginal improvement in throughput, it is desirable to be consistent across all of the systems. In addition, unless the gain in throughput is large, increasing the number of threads increases the elapsed time required to process each query by the same factor. Hence, we chose to use $t = 32$ threads in all of the subsequent experiments.

Table 4 Normalized throughput with various threading levels t, using $k = 8$ nodes, and collection `TB/01`. Throughput is presented in units of terabyte queries per machine second, as discussed in the text

Simultaneous queries, t	Method		
	Document	Term	Pipelined
1	2.08	0.57	0.59
2	3.48	1.09	1.14
4	4.25	1.31	1.99
8	4.70	1.35	2.97
16	4.99	1.37	3.78
32	5.34	1.37	4.10
64	5.22	1.33	4.16

Indeed, there is a direct relation between throughput, number of threads, and average query response time. Throughput is defined as "(queries $\times$ terabytes)/(machines $\times$ elapsed seconds)", from which it follows that "average response time in seconds $=$ (threads $\times$ terabytes) $\times$ (throughput $\times$ machines)". For a given configuration, machines and terabytes remain constant. Thus, if the number of threads is doubled without any increase in throughput, the average query response time also doubles, where response time is measured from the moment a query begins executing on an available thread until the time it completes. So, for example, the normalized throughput of 4.10 for document distribution with 32 threads equates to an average query response time of $(32 \times 0.43)/(4.10 \times 8) = 0.42$ s, whereas the normalized throughput of 4.16 with 64 threads gives an average query response time of 0.83 s. Our approach in this paper is to concentrate primarily on query throughput; a production system would also need to take into account average response time.

5 Experiments with distribution

This section establishes baseline results for monolithic runs, then describes the outcomes of experiments with document-partitioned, term-partitioned, and pipelined querying.

5.1 Monolithic baseline

The first experiments executed the 10,000 queries against monolithic indexes of the different collections, to establish baseline throughput results against which the three distribution models can be compared. Results are presented in Table 5. In order to make the results consistent, the monolithic system is also implemented using a receptionist/processor model, with just a single processor. Thus, the "monolithic" system can be regarded as both a single-processor document-partitioned system and a single-processor pipelined system. Table 5 also shows the total amount of data read off disk (not including data supplied from in-memory file buffers), and the proportion of the elapsed querying time spent waiting for I/O operations to complete. As in all of the tables in this paper, query throughput is normalized by collection size. If the collection size were to be doubled, and the number of queries processed per second had halved as a result, the same normalized throughput figure would be attained.

The throughput results in Table 5 demonstrate that significant efficiencies of scale are possible while the index remains small enough to fit into memory (`TB/64` to `TB/32`); and even when some of it is disk-resident (`TB/16`), further gains are possible. However, once the

Table 5 Throughput and read behavior as a function of collection size, using a single processor, with at most $t = 32$ queries concurrently active. Throughput is presented in units of terabyte queries per machine second. The column "data read" measures data physically read off disk, and does not include any read requests supplied from in-memory buffers. The column "I/O wait load" is the percentage of the total query stream execution time that each server spent doing nothing except waiting for I/O to complete

Collection	Normalized throughput	Data read (GB)	I/O wait load %
`TB/64`	3.18	0.00	0.0
`TB/32`	4.25	0.00	0.0
`TB/16`	5.15	0.10	0.0
`TB/08`	5.41	0.38	4.7
`TB/04`	5.75	1.23	6.1
`TB/02`	6.67	3.33	5.5
`TB/01`	6.83	9.37	6.0

collection has reached the one-eighth size of `TB/08` (with an index of 2.4 GB, see Table 2) physical index reads from disk slow processing, and around 5% of time is spent waiting for disk transfers. The increase in normalized throughput slows in this middle section of the table. Finally, once the penalty of moving from memory-based to disk-based indexing has been passed, normalized throughput sees greater gains (`TB/02`) and then eventual saturation (`TB/01`). The 2 MB per-query average index volume in `TB/01` (Table 2) is twice the size of the 1 MB memory cache.

The total amount of inverted list data processed during the 10,000 queries is 20 GB. Table 5 shows that in the `TB/01` experiment almost half of all data is read off disk as it is required, rather than supplied from in-memory buffers. Once the index no longer fits into the available main memory, I/O wait load remains a steady fraction of elapsed processing time as the collection grows.

5.2 Document partitioning

To test the document-partitioned system, each collection was processed on varying numbers of machines, using the same set of 10,000 queries. Each server built an index for its assigned sub-collection, including formation of a local vocabulary. All queries were executed with global term statistics, based on an aggregate vocabulary maintained by the receptionist that covered all sub-collections. (If a central vocabulary were not available, the servers could be queried by the receptionist to obtain the set of local term weights, and then the aggregate weights broadcast back to the servers, with little alteration in overall querying costs.) Each server then executed the query using the global weights, and returned to the receptionist the identifiers and scores of the top $r' = 1{,}000$ documents in its sub-collection.

Table 6 is the first of three with the same structure, each of which records the results achieved for one of the three different distributed retrieval systems. Each entry in the table is a normalized throughput rate, expressed in terms of terabyte queries per machine second, so that all numbers are directly comparable in a "bang per buck" sense. To evaluate the extent to which collection growth can be managed via distribution, sets of values on a down-to-the-right diagonal line should be considered. For example, starting at the ($k = 1$, `TB/08`) entry, the next interesting combination is ($k = 2$, `TB/04`), in which twice the volume of data is being processed on twice as many machines, and hence with the same volume of data on each machine. Similarly, the ($k = 4$, `TB/02`) entry in Table 6 reflects a system with

Table 6 Normalized throughput rates for document-partitioned distributed retrieval. All values are in units of terabyte queries per machine second. The first row shows the corresponding values for a monolithic configuration. As many as $t = 32$ queries were concurrently active. The receptionist is not counted as one of the processors. Values shown are means over five runs

	Collection						
k	`TB/64`	`TB/32`	`TB/16`	`TB/08`	`TB/04`	`TB/02`	`TB/01`
1	3.18	4.25	5.15	5.41	5.75	6.67	6.83
2	–	3.18	4.29	5.11	5.48	5.94	6.62
4	–	–	3.13	4.25	5.11	5.42	5.89
8	–	–	–	2.91	4.15	5.08	5.34

four times as much data on four times as many machines as the ($k = 1$, `TB/08`) entry. Any down-to-the-right diagonal line in the table allows a similar exploration of scalability.

The general pattern down these diagonals is consistent—as the volume of data grows in proportion to the number of machines, normalized throughput tends to decrease slightly as a result of two factors. First, the greater the number of partitions, the greater the network overhead. This is particularly the case for 8-way `TB/08` and `TB/04`, where the nodes finish each query so rapidly that the receptionist is unable to dispatch queries quickly enough to keep them busy. The second reason is the synchronized way in which the document-partitioned system works. Since each query goes to every machine, the time taken to evaluate it is determined by the slowest machine. Although document partitioning does very well at evenly distributing the load, the system as a whole tends to run at the speed of the slowest node. The more nodes that are added to the system, the greater the variance between the average and worst-performing node, and (marginally) the slower the system becomes.

Another way of interpreting Table 6 is to look vertically down a column, which shows what happens when the same volume of data is split into smaller parts and spread across an increasing number of servers. The general trend down columns is for normalized throughput to slightly decrease. That is, the system is most efficient when all the data is on a single machine, and communication overheads mean that doubling the number of machines from k to $2k$ does not double the number of queries per second the system as a whole can handle, even though it (nearly) halves the average response time. Nevertheless, document partitioning does scale reasonably well with cluster size.

5.3 Term partitioning

Table 7 shows the result of carrying out the same experiment with a term-partitioned arrangement. Terms were assigned to servers arbitrarily, using a hashing technique, and the

Table 7 Normalized throughput rates for term-partitioned distributed retrieval. All values are in units of terabyte queries per machine second. As many as $t = 32$ queries were concurrently active. The receptionist is not counted as one of the processors. Values shown are means over five runs

	Collection						
k	`TB/64`	`TB/32`	`TB/16`	`TB/08`	`TB/04`	`TB/02`	`TB/01`
1	–	–	–	–	–	–	–
2	–	2.46	3.32	4.13	4.74	5.22	5.47
4	–	–	1.70	2.08	2.37	2.58	2.75
8	–	–	–	1.03	1.18	1.29	1.37

partitioning was not influenced by any analysis of term frequency either in the collection or in the query set.

Once a query had been parsed by the receptionist, requests were issued for the required inverted lists, using a vocabulary-division table maintained by the receptionist. Complete inverted lists were returned from each of the servers to the receptionist. When all of the required inverted lists were back from the servers, the receptionist evaluated the query and determined the top-ranked $r = 1{,}000$ documents.

As can be seen Table 7, term partitioning derives almost no advantage from distribution, and throughput is only a small fraction of that achieved with document partitioning. The columns in the table highlight how wasteful this approach is—as the number of processors is doubled and the volume of data is held the same, the actual time taken to perform the query run remains roughly constant, and normalized throughput halves. That is, term-partitioning fails to scale at all.

Part of the problem is that the receptionist becomes a bottle-neck, and in all except the smallest collections, spends roughly 98% of its time in a non-idle state. The receptionist is essentially uni-processing all of the queries, and although the other machines are serving index lists, they are largely idle. The pattern of results in Table 7 confirm the observations of MacFarlane et al. (2000), while showing that the results of Badue et al. (2001) do not generalize to large numbers of real queries. The other aspect of the term-partitioned arrangement that creates problems is network load: recorded network traffic was 0.43 Gbit per second for the `TB/01` runs, close to the practical limit of the network connecting the cluster. In contrast, the volume of network traffic generated by the document-distributed arrangement is extremely small—no more than a few tens of kilobytes per query.

In addition, the methodology by which costs are normalized by the number of servers significantly overestimates the true throughput of term distribution. In a document-partitioned system the receptionist does relatively little work, and could easily share a machine with a server; thus, it is reasonable to not count in the arithmetic the processor that is required to support the receptionist. In term-distribution, however, the receptionist is a key bottleneck. If the receptionist were properly counted as an extra machine the already poor results reported for this approach would be further lowered, and the seemingly attractive throughput rates in the $k = 2$ row would be reduced by a third.

5.4 Pipelining

Table 8 shows the corresponding throughput results for the new pipelined evaluation approach. There are two patterns to be noted. First, going down the "equal work per processor" diagonals, $k = 2$ pipelining generally outperforms monolithic processing for the same volume

Table 8 Normalized throughput rate for pipelined query evaluation with a term-partitioned index. All values are in units of terabyte queries per machine second. The first row shows the corresponding values for a monolithic configuration. As many as $t = 32$ queries were concurrently active. The receptionist is not counted as one of the processors. Values shown are means over five runs

	Collection						
k	`TB/64`	`TB/32`	`TB/16`	`TB/08`	`TB/04`	`TB/02`	`TB/01`
1	3.18	4.25	5.15	5.41	5.75	6.67	6.83
2	–	3.72	4.54	5.18	5.58	5.98	6.37
4	–	–	3.88	4.45	4.92	5.25	5.53
8	–	–	–	3.38	3.67	3.95	4.10

of data per node. However, $k=4$ pipelining has similar normalized throughput to the comparable $k=1$ monolithic system, and $k=8$ pipelining has worse.

Second, going down the columns, normalized throughput falls off sharply as machines are added to the system. Comparing pipelining's throughput rates with those for document partitioning in Table 6 reinforces this pattern. Pipelining does not scale as well as does document partitioning. For $k=8$ partitioning of `TB/01`, pipelining is only able to achieve slightly over 75% of the normalized throughput of the document-partitioned system.

Pipelining is clearly to be preferred as an architecture if querying must be supported on top of a term-partitioned index. It gives much better throughput figures than term partitioning. It also uses significantly less network bandwidth, recording an average rate of 0.10 Gbit per second for the $k=8$ `TB/01` run, compared to 0.43 for term-partitioning.

Nevertheless, on the test data and queries, it is unable to match the document-partitioned system. Either pipelining's anticipated superiority in disk reads and memory usage do not eventuate in reality, or they are outweighed by problems with workload balancing. Section 6 discusses these issues.

5.5 Tuning

Section 4 proposed an accumulator limit of 100,000, scaled with collection size, as offering the best tradeoff between retrieval effectiveness and memory efficiency. Since the pipelined system is required to ship accumulator bundles between nodes, it should suffer degraded performance from a greater number of accumulators, and improved performance from a lessened one, both in absolute terms and when compared with document partitioning. To test the extent to which the accumulator limit was a factor in the measured performance, the $k=8$ `TB/01` configurations for each of the distribution methods were compared, varying the number of accumulators allowed during the similarity computation. Table 9 shows the outcome of these experiments.

As expected, an increase in the number of permitted accumulators reduces the query throughput that can be achieved. Performance degradation is heaviest for pipelining, but occurs for all distribution methods. Document partitioning's throughput falls roughly 35% as the accumulator limit grows from 40,000 to 400,000, whereas pipelining's falls roughly 50%, suggesting that the additional cost of shipping accumulator bundles, borne by pipelining alone, is half as significant as the additional memory costs of the greater number of accumulators, borne by all methods. Note that even with 400,000 accumulators, the pipelined

Table 9 Normalized throughput rates, measured in terabyte queries per machine second, as the number of accumulators is varied. In each experiment $k=8$ processors were applied to the `TB/01` collection. All values are in units of terabyte queries per machine second. As many as $t=32$ queries were concurrently active. The receptionist is not counted as one of the processors. Values shown are means over five runs

	Method		
Accumulators	Document	Term	Pipelined
40,000	5.83	1.64	5.03
100,000	5.34	1.37	4.10
200,000	4.34	1.13	3.34
400,000	3.73	0.88	2.48

Table 10 Read behavior of document-partitioned and pipelined distributed systems, $k = 8$ nodes, and `TB/01`. Figures are totaled across all nodes. Disk sectors are 512 bytes. Numbers reflect physical disk activity only, and requests handled from in-memory buffers are not included in the totals

	Method	
Statistic	Document	Pipelined
Sectors read ($\times 10^6$)	6.61	2.60
Distinct reads ($\times 10^4$)	10.27	1.46
I/O wait seconds ($\times 10^1$)	13.85	3.28

system is still well within the bandwidth limits of the network; network data shows an average cluster-wide network traffic of 0.13 Gbit per second.

6 Workload distribution

Our opening hypothesis was that the pipelined method would gain a performance advantage over document partitioning due to its better memory usage and consolidated representation of index data on disk. Table 10 demonstrates that pipelining does indeed have these advantages. For a $k = 8$-way run on `TB/01` run, the pipelined system fetched only 40% as much data from disk as the document-partitioned one, and took just 15% as many disk read operations to do it, indicating better caching effectiveness and reduced fragmentation of data. As a result, the pipelined system spent only 25% as much time blocked waiting on I/O as the document-partitioned system did.

Despite these promising relativities, the throughput figures in Tables 6 and 8 indicate that overall, pipelining's performance is at best only comparable with document partitioning. Moreover, pipelined distribution, as described here, scales poorly with cluster size.

As part of the explanation for the discrepancy between disk operations and overall throughput, Fig. 3 shows the "busy load" at each node in an $k = 8$-node run on `TB/01` over the duration of the 10,000 queries, for the document-partitioned and pipelined distribution methods. Busy load is defined here as the proportion of the time a node spends doing anything

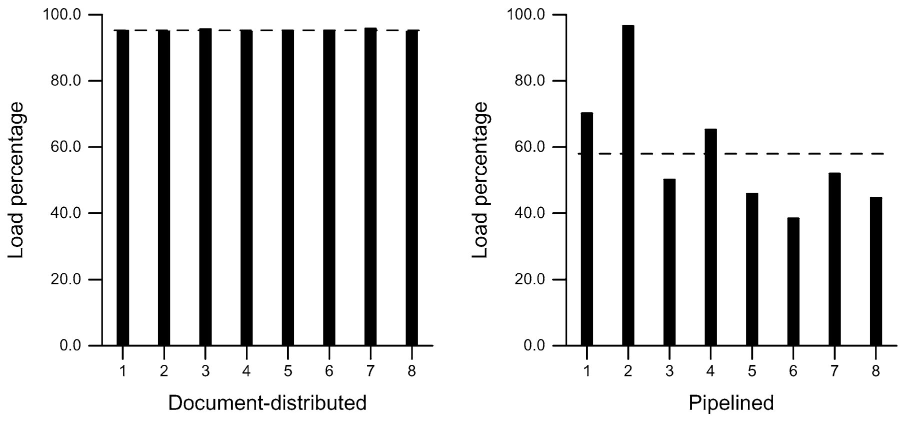


Fig. 3 Average per-processor busy load for $k = 8$ and `TB/01`, for document-distributed processing and pipelined processing. The dashed line in each graph is the average busy load over the eight processors

except the idle task; in particular, it includes time spent waiting for I/O to complete. The left-hand graph shows that eight nodes in the document-partitioned have very similar busy loads, indicating an even distribution of the processing workload. On the other hand, the right-hand graph shows that nodes in the pipelined system have uneven busy loads, with the most idle node being busy less than 40% of the time. Node two, busy on average 95% through the query sequence, is a bottleneck for the system as a whole. The other nodes become starved for work as queries with a term on node two queue up for processing.

The document-partitioned system has a much higher average busy load than the partitioned one, 95.3% compared to 58.0%. On the one hand, this is to the credit of document-distribution, in that it demonstrates that it is better able to make use of all system resources, whereas pipelining leaves the system underutilized. On the other hand, the fact that in this configuration pipelining is able to achieve roughly 75% of the throughput of document-distribution using only 60% of the resources is encouraging, and confirms the model's underlying potential.

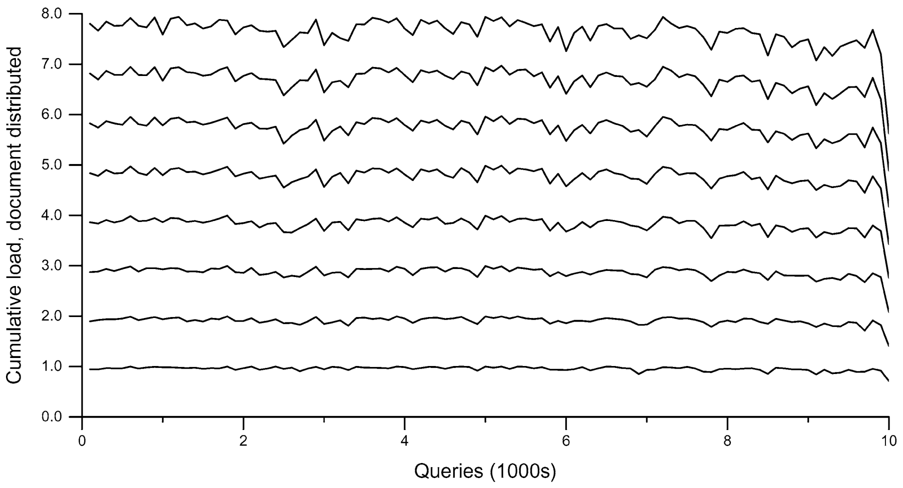
Figure 3 summarized system load for the 10,000-query run as a whole; it is also instructive to examine the load over shorter intervals. Figure 4 shows the busy load for document-distribution (top) and pipelining (bottom) when measured every 100 queries, with the eight lines in each graph showing the aggregate load of this and lower numbered processors. That is, the eighth of the lines shows the overall total as a system load out of the available 8.0 total resource. Note how the document distributed approach is consistent in its performance over all time intervals, and the total system utilization remains in a band between 7.1 and 7.8 out of 8.0 (ignoring the trail-off as the system is finishing up).

The contrast with pipelining (the bottom graph in Fig. 4) is stark. The total system utilization varies between 3.1 and 5.9 out of 8.0, and is volatile, in that nodes are not all busy or all quiet at the same time. The only constant is that node two is busy all the time, acting as a throttle on the performance of the system.

The reason for the uneven system load lies in the different ways the document-partitioned and pipelined systems split the collection. The two chief determinants of system load in a text query evaluation engine are the number of terms to be processed, and the length of each term's inverted lists. Document partitioning divides up this load evenly—each node processes every term in the query, and the inverted list for each term is split approximately evenly among the nodes. Variation is introduced only if term usage is not homogeneous across the collection.

In the pipelined system the work load of any single query is potentially divided quite unevenly between the nodes. For example, it is possible that all terms in the query are handled by a single node; and even if no node is hit by more than one term, the length of the inverted list for different terms can vary greatly. The weight of data generated by processing many queries in parallel should dilute this effect; nevertheless, the arbitrary nature of query input means that at any given time, some nodes are likely to be more heavily loaded, while others are relatively idle.

Term frequencies tend to follow a power-law distribution, meaning that frequency is highly skewed to the most frequent terms. Being frequent in the collection gives a term a long inverted list; and being frequent in the query stream means that the list will be used often. Multiplying a term's frequency in the query set by the length of the term's inverted list in the index thus gives an estimate of the processing load arising from that term. For the synthetic query set and collection used in these experiments, the highest term processing load takes up 6.6% of the total query set processing load, and has 2.2 times the processing load of the second highest term, and 3.5 times that of the third. Given the highly skewed nature of term processing loads, a random distribution of terms amongst nodes (which is



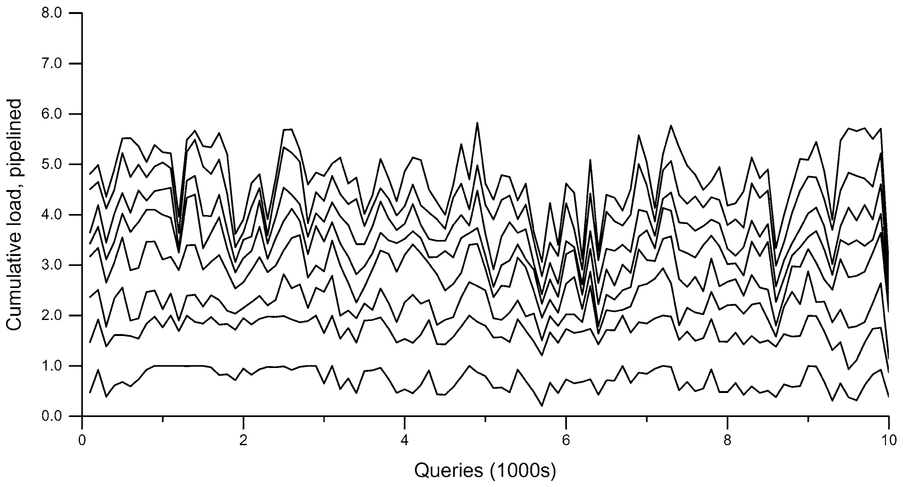


Fig. 4 Aggregate load over the eight processors in a $k = 8$ run on `TB/01`. The eight lines in the top graph reflect the pointwise aggregate load utilization of the eight processors in a document-distributed system; the eight lines in the lower graph reflect the pointwise aggregate load utilization of the eight processors in an equivalent pipelined system. Load is measured every 100 queries in a 10,000 query run

what our hash-based term partitioning scheme hopes to approximate) is likely to result in an uneven node processing load.

Figure 5 shows the total list processing load on each node in a $k = 8$-node run on `TB/01`. In the left-hand graph, document partitioning divides the load almost perfectly evenly, and all nodes process an equal amount of the index data. Pipelining is again uneven, with the most heavily weighted node (node two) have almost three times the processing load of the most lightly weighted one. As it turns out, the hashing algorithm used for term assignments to nodes placed the first, third, and fourth heaviest terms onto this one node; and while it might be tempting to dismiss this as bad luck and choose a different hash, the properties of the query stream cannot be known when the system is partitioned, and any random process is likely to run into similar problems. The problem becomes worse as the number of partitions increase—at sixteen partitions, for example, the dominant term would need to have a node to

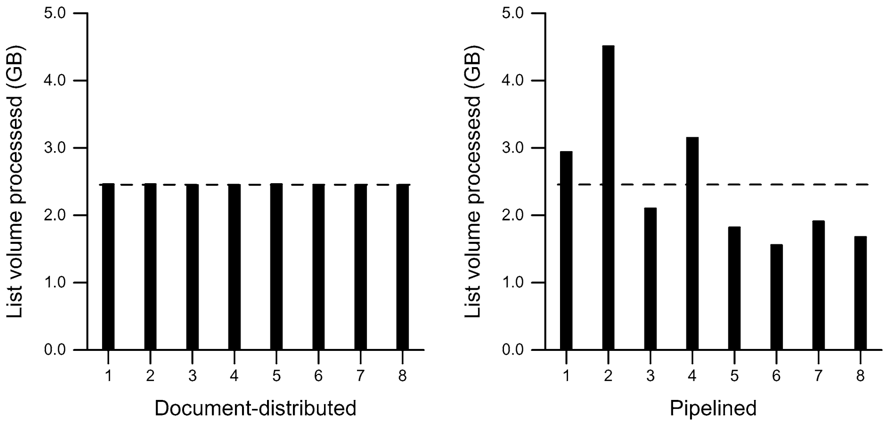


Fig. 5 Total inverted list lengths processed on each node, in GB, for a $k = 8$-node run on `TB/01`. Both systems process the same total volume of index data, but share the work differently

itself if processing load is to be balanced. On the other hand, the problem of load imbalance would likely be reduced if answers to frequently asked queries were cached, as is commonly done in production systems, since frequent queries are likely to contain frequent words. This is an area that requires further investigation.

7 Future directions

To establish how well pipelining might perform if the workload between nodes was more balanced, the synthetic query generation process was modified to generate a more benign set of queries. The same vocabulary of translated terms was used as before, but instead of mapping the original queries term-by-term based on relative frequency, queries were generated by randomly choosing terms from the vocabulary. The same distribution of query lengths was maintained, but the distribution of query term frequencies became much more even, removing one of the two factors causing skew in term processing load. The unevenness caused by differing inverted list lengths was left unchanged.

The modified query log was again executed against $k = 8$-node document-partitioned and pipelined indexes of `TB/01`. Removing the query set term frequency skew leads to a lower overall processing load; the total size of inverted lists processed for the 10,000 query run fell from 19.7 GB to 8.7 GB. Not surprising, therefore, was that throughput of the document-partitioned system rose from 5.34 to 6.83 terabyte queries per machine second. But the improvement in the throughput of pipelining was far more dramatic. With the original query set, throughput was 4.10 terabyte queries per machine second, as shown in Table 8. With the random query translation, this doubled, to 8.29 terabyte queries per machine second, over 20% more than document-partitioned. Different term inverted list lengths meant that the busiest node processed 1.5 GB of data compared to the lightest load of 1.0 GB, so further balancing might yield additional gains.

These positive results cannot, of course, be claimed as improvements, since they involve an atypical query stream. But they do offer encouragement, in that if a way of balancing the processing requirements of the pipelined scheme can be found, it should be possible to fully crystallize the benefits generated by its inherent advantages in terms of disk operations.

Recent work has explored both dynamic load balancing and reassignment of lists while the query stream is being processed, and also selective list replication (Moffat et al., 2006).

It is also worth noting that even 426 GB is not an especially large document collection by web standards. Further experimentation on larger data sets will be pursued as part of the ongoing investigation.

8 Conclusions

We have undertaken a detailed experimental comparison of existing approaches to distributing query evaluation in a cluster, and compared them to a new pipelined approach. Key features of our results are that they are based on actual experiment rather than simulation; make use of a large volume of data rather than having been extrapolated from small-scale experiments; are based on a long stream of realistic queries rather than on small, inappropriate, or simplisticly simulated query sets; and explore several crucial implementation issues, such as the use of concurrent query threads to ensure that peak throughput is achieved and resources are not unnecessarily idle. We have also provided detailed instrumentation of the testing, so as to accurately identify the various costs that make up the overall processing time in a distributed system.

Our experiments have demonstrated that the new pipelined approach to distribution offers much better performance on top of a term-partitioned index than the traditional term-partitioned evaluation method, while still maintaining the latter's desirable I/O properties. The drawback of the new method is that poor balancing of workload means that it fails to outperform document distribution, and the workload balance issues become more serious as the degree of distribution increases.

The new method does have some advantages over document distribution. It makes better use of the memory resources available on the nodes, and requires significantly fewer disk seeks and transfers. These desirable attributes mean that further work to address the load balancing problems—perhaps via selective list replication or dynamic list redistribution—may lead to a system capable of out-performing document distribution.

Acknowledgment Bruce Croft (University of Massachusetts) provided helpful input.

References

Anh, V. N., de Kretser, O., & Moffat, A. (2001). Vector-space ranking with effective early termination. In W. B. Croft, D. J. Harper, D. H. Kraft, and J. Zobel (Eds.), *Proc. 24th Annual International ACM SIGIR Conf. on Research and Development in Information Retrieval* (pp. 35–42), New Orleans, LA. New York: ACM Press.

Anh, V. N., & Moffat, A. (2006). Pruned query evaluation using pre-computed impacts. In S. Dumais, E. N. Efthimiadis, D. Hawking and K. Järvelin (Eds.), *Proc. 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 372–379). Seattle, Washington. New York: ACM Press. URL http://doi.acm.org/10.1145/1148170.1148235.

Badue, C., Baeza-Yates, R., Ribeiro-Neto, B., & Ziviani, N. (2001). Distributed query processing using partitioned inverted files. In G. Navarro (Ed.), *Proc. Symp. String Processing and Information Retrieval* (pp. 10–20). Laguna de San Rafael, Chile.

Baeza-Yates, R., & Ribeiro-Neto, B. (1999). *Modern information retrieval*. New York, NY: ACM Press.

Bailey, P., Craswell, N., & Hawking, D. (2003). Engineering a multi-purpose test collection for web retrieval experiments. *Information Processing & Management*, *39*(6), 853–871.

Barroso, L. A., Dean, J., & Hölzle, U. (2003). Web search for a planet: The Google cluster architecture. *IEEE Micro*, *23*(2), 22–28.

Cacheda, F., Plachouras, V., & Ounis, I. (2004). Performance analysis of distributed architectures to index one terabyte of text. In S. McDonald and J. Tait (Eds.), *Proc. 26th European Conf. on IR Research, volume 2997 of Lecture Notes in Computer Science* (pp. 394–408), Sunderland, UK. Springer.

Cahoon, B., McKinley, K. S., & Lu, Z. (2000). Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *ACM Transactions on Information Systems*, *18*(1), 1–43.

Clarke, C. L. A., Tilker, P. L., Tran, A. Q.-L., Harris, K., & Cheng, A. S. (2003). A reliable storage management layer for distributed information retrieval systems. In *Proc. 2003 CIKM Int. Conf. Information and Knowledge Management* (pp. 207–215), New York, NY, USA. New York: ACM Press.

de Kretser, O., Moffat, A., Shimmin, T., & Zobel, J. (1998). Methodologies for distributed information retrieval. In M. P. Papazoglou, M. Takizawa, B. Krämer, and S. Chanson (Eds.), *Proc. 18th International Conf. on Distributed Computing Systems* (pp. 66–73), Amsterdam, The Netherlands, IEEE.

Harman, D., McCoy, W., Toense, R., & Candela, G. (1991). Prototyping a distributed information retrieval system using statistical ranking. *Information Processing & Management*, *27*(5), 449–460.

Hawking, D. (1998). Efficiency/effectiveness trade-offs in query processing. *ACM SIGIR Forum*, *32*(2), 16–22.

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

Lester, N., Moffat, A., Webber, W., & Zobel, J. (2005a). Space-limited ranked query evaluation using adaptive pruning. In A. H. H. Ngu, M. Kitsuregawa, E. J. Neuhold, J.-Y. Chung, and Q. Z. Sheng (Eds.), *Proc. 6th International Conf. on Web Information Systems Engineering* (pp. 470–477), New York, LNCS 3806, Springer.

Lester, N., Moffat, A., & Zobel, J. (2005b). Fast on-line index construction by geometric partitioning. In A. Chowdhury, N. Fuhr, M. Ronthaler, H.-J. Schek, and W. Teiken (Eds.), *Proc. 2005 CIKM Int. Conf. Information and Knowledge Management* (pp. 776–783), Bremen, Germany, New York: ACM Press.

MacFarlane, A., McCann, J. A., & Robertson, S. E. (2000). Parallel search using partitioned inverted files. In P. de la Fuente (Ed.), *Proc. Symp. String Processing and Information Retrieval* (pp. 209–220) A Coruña, Spain.

Moffat, A., Webber, W., & Zobel, J. (2006). Load balancing for term-distributed parallel retrieval. In S. Dumais, E. N. Efthimiadis, D. Hawking and K. Järvelin (Eds.), *Proc. 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 348–355). Seattle, Washington. New York: ACM Press. URL http://doi.acm.org/10.1145/1148170.1148232.

Moffat, A., & Zobel, J. (1996). Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, *14*(4), 349–379.

Moffat, A., & Zobel, J. (2004). What does it mean to "measure performance"? In X. Zhou, S. Su, M. P. Papazoglou, M. E. Owlowska, and K. Jeffrey (Eds.), *Proc. 5th Int. Conf. on Web Informations Systems* (pp. 1–12), Brisbane, Australia. LNCS 3306, Springer.

Orlando, S., Perego, R., & Silvestri, F. (2001). Design of a parallel and distributed web search engine. In *Proc. 2001 Parallel Computing Conf.* (pp. 197–204), Naples, Italy, Imperial College Press.

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

Ribeiro-Neto, B., de Moura, E. S., Neubert, M. S., & Ziviani, N. (1999). Efficient distributed algorithms to build inverted files. In M. Hearst, F. Gey, and R. Tong (Eds.), *Proc. 22nd Annual International ACM SIGIR Conf. on Research and Development in Information Retrieval* (pp. 105–112), San Francisco, CA, New York: ACM Press.

Ribeiro-Neto, B. A., & Barbosa, R. R. (1998). Query performance for tightly coupled distributed digital libraries. In *Proc. 3rd ACM Conf. on Digital Libraries* (pp. 182–190), Pittsburgh, PA, New York: ACM Press.

Scholer, F., Williams, H. E., Yiannis, J., & Zobel, J. (2002). Compression of inverted indexes for fast query evaluation. In M. Beaulieu, R. Baeza-Yates, S. H. Myaeng, and K. Järvelin (Eds.), *Proc. 25th Annual International ACM SIGIR Conf. on Research and Development in Information Retrieval* (pp. 222–229), Tampere, Finland, New York: ACM Press.

Sornil, O. (2001). *Parallel inverted index for large-scale, dynamic digital libraries*. PhD thesis, Virginia Tech., USA.

Spink, A., Wolfram, D., Jansen, B. J., & Saracevic, T. (2001). Searching the web: The public and their queries. *Journal of the American Society for Information Science*, *52*(3), 226–234.

Tomasic, A., & García-Molina, H. (1993). Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In M. J. Carey and P. Valduriez (Eds.), *Proc. 2nd International Conf. On Parallel and Distributed Information Systems* (pp. 8–17), Los Alamitos, CA, IEEE Computer Society Press.

Webber, W., & Moffat, A. (2005). In search of reliable retrieval experiments. In J. Kay, A. Turpin, and R. Wilkinson (Eds.), *Proc. 10th Australasian Document Computing Symposium* (pp. 26–33), Sydney, Australia, University of Sydney.

Witten, I. H., Moffat, A., & Bell, T. C. (1999). *Managing gigabytes: Compressing and indexing documents and images*, 2nd ed. San Francisco, Morgan Kaufmann.

Xi, W., Sornil, O., Luo, M., & Fox, E. A. (2002). Hybrid partition inverted files: Experimental validation. In M. Agosti and C. Thanos (Eds.), *Proc. European Conf. on Digital Libraries* (pp. 422–431), Rome. LNCS volume 2458, Springer.

Zhai, C., & Lafferty, J. (2004). A study of smoothing methods for language models applied to information retrieval. *ACM Transactions on Information Systems*, *22*(2), 179–214.

Zobel, J., & Moffat, A. (2006). Inverted files for text search engines. *ACM Computing Surveys*, *38*(2). URL http://doi.acm.org/10.1145/1132956.1132959.