Load Balancing for Term-Distributed Parallel Retrieval

Alistair Moffat
Computer Science and
Software Engineering
The University of Melbourne
Victoria 3010, Australia
alistair@csse.unimelb.edu.au

William Webber
Computer Science and
Software Engineering
The University of Melbourne
Victoria 3010, Australia
wew@csse.unimelb.edu.au

Justin Zobel
Computer Science and
Information Technology
RMIT University
Victoria 3001, Australia
jz@cs.rmit.edu.au

ABSTRACT

Large-scale web and text retrieval systems deal with amounts of data that greatly exceed the capacity of any single machine. To handle the necessary data volumes and query throughput rates, parallel systems are used, in which the document and index data are split across tightly-clustered distributed computing systems. The index data can be distributed either by document or by term. In this paper we examine methods for load balancing in term-distributed parallel architectures, and propose a suite of techniques for reducing net querying costs. In combination, the techniques we describe allow a 30% improvement in query throughput when tested on an eight-node parallel computer system.

Categories and Subject Descriptors

H.3.1 [Information Storage and Retrieval]: Content analysis and indexing – *indexing methods*; H.3.2 [Information Storage and Retrieval]: Information storage – *file organization*; H.3.3 [Information Storage and Retrieval]: Information search and retrieval – *search process*; H.3.4 [Information Storage and Retrieval]: Systems and software – *performance evaluation*.

General Terms

Efficiency, performance, algorithms.

1. INTRODUCTION

The amount of data that must be handled by a large-scale information retrieval system greatly exceeds the capacity of any single machine. For example, web search engines manage collections measured in the terabyte range; if this volume of data were to be stored and indexed on a single computer, queries would take many tens of seconds to evaluate with even the most efficient index representations and query resolution methods.

To handle the necessary data volumes and query throughput rates, parallel systems are used, in which the document and index data are split across tightly-clustered distributed computing systems. For example, in a *document-distributed* system, each processing node stores the index corresponding to a subset of the documents; queries are processed in parallel at all nodes; and collated back into a single combined answer when all nodes have completed their local processing. Every node participates in the resolution of every query.

An alternative is to use a *term-distributed* index, in which each of the processing nodes maintains complete index information for a subset of the terms in the collection, and each query is referred to the subset of the nodes that hold relevant information. In a standard term-distributed query resolution method, a receptionist receives a query, requests the index information for the query terms from the pertinent nodes, and processes this information centrally. Compared to document-distributed parallelism, such term-distributed indexing has the drawback of a severe bottleneck at the receptionist. An alternative is the *pipelined* term-distributed evaluation strategy of Moffat et al. [2005], where the query processing is distributed across the nodes. Document-partitioning achieves more even load balancing than does pipelining, but has the disadvantage of requiring more disk accesses, because the index information for each term is stored across multiple machines.

In this paper we examine methods for load balancing in pipelined term-distributed architectures, and propose a suite of techniques for reducing net querying costs. In particular, we explore the load distribution behavior that pipelining displays, and show that the imbalances can be addressed by techniques that include predictive index list assignments to nodes, and selective index list replication.

In combination, the techniques we describe allow a 30% improvement in query throughput when tested on an eight-node parallel computer system, and result in a term-distributed implementation of parallel querying that approaches the query throughput rates of a document-distributed system, with lower total CPU workloads.

2. DISTRIBUTED RETRIEVAL

In a standard monolithic retrieval system, a collection of documents and an index for the documents are stored on the same server. An efficient representation of an inverted index consists of a vocabulary of indexed terms and, for each term, an inverted list of information about which documents contain the term. The inverted list might also contain information such as within-document frequencies, and word positions in the document. Queries are resolved by fetching the inverted lists corresponding to the query terms, and using the list information to incrementally build a set of *accumulators* that, by the time the last term has been processed, store the similarity of each document to the query. Through careful run-time pruning of documents with low similarity [Lester et al., 2005], the set of accumulators can be kept small, with no more than one accumulator for every 100 indexed documents.

A parallel, or *tightly distributed*, retrieval system is one in which responsibility for the management of the collection of documents

SIGIR'06, August 6–11, 2006, Seattle, Washington, USA.

is partitioned, under centralized control, over multiple computers sharing a high-bandwidth network.

Other types of distributed system include meta-search systems, where queries are processed centrally but the individual document collections are independent; and peer-to-peer systems, in which there is no central coordination, possibly low-bandwidth network connectivity, and unknown amounts of document duplication. We do not consider these kinds of distributed system here. While techniques developed for meta-search, such as collection selection, have some appeal as a way of reducing costs in the document-distributed systems described below, the fact that they are less effective than fully indexed systems make them attractive only when other approaches cannot cope.

Nor do we consider mirroring – building of multiple copies of the complete system so as to correspondingly multiply throughput. Our concern here is with retrieval environments in which there is so much data that storing it all on a single machine would be either impossible, because of hardware limitations; or impractical, because of excessive response times to queries.

Two principal distribution paradigms have been developed. One approach is to have a *document distributed* index and a corresponding query processing regime. In a document-distributed system, the collection is split across the k available processors (or *nodes*), so that each node is responsible for approximately $1/k$ th of the collection. Nodes then build local indexes for those partitions, and answer queries against them. Queries to the system as a whole are routed to all of the k nodes in the network, each of which evaluates the query and returns a set of r results to the coordinating machine (the *receptionist*). The receptionist then collates the rk answers into a final list, and returns the top r of them to the user.

Document distribution has a number of natural advantages, not the least of which is that each machine is in essence managing a smaller monolithic collection – an arrangement for which index construction and query processing mechanisms are well understood. Document distribution also ensures a relatively stable balance of workload across the k machines, since each is processing the same queries, at the same time, at roughly equal cost. In addition, document distribution provides natural support when the documents themselves are to be supplied to the user – they can be retained on the machine that indexed them.

The alternative is *term-distribution*. In a term-distributed index, each processor stores index information about a subset of the terms, rather than a subset of the documents. When the receptionist receives a query it requests index information from the relevant nodes, and combines that information to form a list of overall answers. Only nodes that store information relating to a term in that current query are required to take any action.

The key advantage of term-based distribution is that there are fewer disk accesses, because index information for each term is now stored in just one location. Term-distribution also implies that less main memory is required for storage of redundant vocabulary information, freeing up memory for useful caching; in a document-distributed system, each node must maintain vocabulary information for every term within its partition, and a large number of terms appear in more than one partition.

Term-based distribution does, however, have a major disadvantage – in the simple form described here, the receptionist becomes a bottleneck, and, because the bulk of the actual computation is performed using the index lists, starves the other nodes of useful work. In effect, the nodes become little more than inverted list servers; Moffat et al. [2005] show that standard term distribution is unable to obtain significant gains in throughput when additional machines are made available. Moreover, even if the inverted lists do not contain word positions they can be long, meaning that significant network traffic might be incurred.

Since the nodes are doing little work, while the receptionist is at full load, there is a severe load imbalance. The issue of load balancing is central to our contributions in this paper. The next section describes a pipelined approach to query evaluation in a term-distributed environment, and then Section 4 returns to the issue of load balancing.

3. THE PIPELINED APPROACH

In a term-distributed system, an obvious way to eliminate the bottleneck at the receptionist is to have multiple query evaluators, one at each node. Each node can then request term information as needed from other nodes, and return completely evaluated queries to the receptionist. However, such an approach has two disadvantages. One is that the movement of term lists creates a significant amount of network traffic. The second is that the caching behavior at each node is poor, as memory is used both for the lists held at that node and for lists that have been imported from elsewhere. Our experience with monolithic and document-distributed systems is that effective use of memory as a cache of recently processed lists – there tends to be high temporal locality in use of query terms – is critical to achieving high throughput.

An alternative approach to eliminating the bottleneck is to use *pipelining* [Moffat et al., 2005]. In this approach, the query is evaluated in stages by the sequence of nodes that hold the inverted lists corresponding to the query terms. The key idea is that rather than pass around inverted lists, a partially evaluated query – represented by the set of accumulators, or document-similarity contributions corresponding to the terms that have already been processed – is circulated to the relevant nodes. Each node that receives the query package applies the updates generated by one or more of the query terms, and then passes the package to the next addressee on the routing list that accompanies it.

In the pipelined approach, the query evaluation strategy is term-by-term. For example, suppose that there are $k = 8$ nodes, and that a query has four terms, t_1, t_2, t_3, and t_4, with the inverted lists held on nodes A (t_1), B (t_2 and t_3), and C (t_4). Evaluation of the query begins on node A, which processes the list corresponding to term t_1 to produce an initial set of accumulators. This set is passed to node B, which processes the lists for t_2 and t_3 against these accumulators to produce a modified set. The modified set is passed to node C, which applies the updates generated by the index list for t_4 to produce a final set of accumulators, and from them extracts the top r answers to return to the receptionist. The only work the receptionist need do is receive each query, plan its

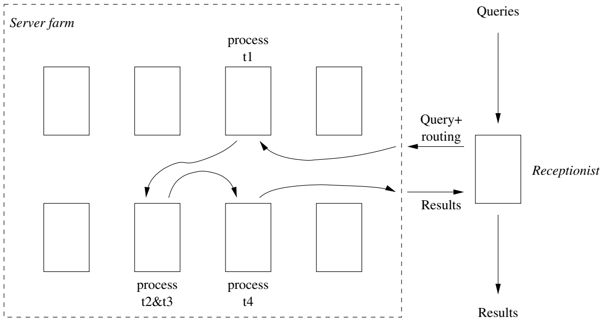

Figure 1: Example of pipelined query evaluation.

path through the nodes, and return the answer lists to the user, as is shown in Figure 1.

In pipelining, each node needs to have enough memory set aside for accumulators for each of a fixed number of threads (the number of simultaneous threads needs to be capped to a reasonable level to prevent thrashing), so the impact on caching is lower than the first version of distributed query evaluation outlined above. The network traffic is limited to the accumulator structures that are passed around between nodes.

However, experiments with pipelining revealed shortcomings in the method [Webber and Moffat, 2005, Moffat et al., 2005]. The most acute was load balance – a small number of query terms contributed a significant fraction of the total workload. These were not necessarily the most common terms in the collection, but rather, were terms whose product of their frequency in the collection and their frequency in the query stream meant that a disproportionate proportion of total processing effort was being spent on them. The nodes that held the index information for these high-workload terms were busier than the remainder, leading to load imbalance. Over short sequences of queries, this problem could be exacerbated by phenomena such as locally high repetition of particular query terms. While these problems could potentially be ameliorated by, for example, caching answers to recent queries, even without repetition of queries the problem would remain.

In the next section we describe alternative approaches to load balancing in a pipelined distributed retrieval system, and in the subsequent section report on our experiments measuring the impact of these approaches.

4. LOAD BALANCING

In their summary of the pipelined approach, Moffat et al. [2005] conclude that a lack of natural load balancing in the pipelined approach is a serious handicap, and that a random assignment of terms to processing nodes risks serious bottlenecks emerging. In particular, if two relatively high workload terms are placed on the same machine, then it is hard for other machines to operate at full efficiency, and overall throughput suffers.

Moffat et al. define workload as follows: for a term t that appears in a query batch Q_t times, and has an inverted list that is B_t bytes long (using some appropriate representation, including compression), the workload L_t associated with that term in that batch is given by

$$L_t = Q_t \times B_t \,.$$

The workload associated with a processing node is the sum of the workloads of the terms assigned to that node. That is, the workload of a node is the total length of compressed inverted lists that must be processed at that node during the execution of a query stream.

To investigate the problem of load balance, we used a version of the Zettair search engine[1] to index the 426 GB GOV2 crawl of the .gov domain used in the TREC Terabyte Track since 2004. All the experiments in this paper are on this data. Zettair was modified by us to support document-distributed and pipelined distributed retrieval; with a necessary part of the modification being the addition of multi-threaded query evaluation. For these initial experiments with load, we used a set SYNQ of queries that have been artificially adapted to the GOV2 crawl to give term-frequency, repetition, and answer-frequency properties close to those of real queries (the Excite97 query log) on general web data (the TREC wt10g collection) [Webber and Moffat, 2005].

Table 1 illustrates the nature of the load balancing problem. To prepare the table, a set of 60,000 SYNQ queries was broken into six batches each of 10,000. A random assignment of term lists to processors was effected via a hash function, and then the workloads of the query batches evaluated with respect to the term assignment, by simply calculating values for L_t and then summing them, without actually answering the queries. For each query batch, the node assigned the heaviest workload in this simulated environment was normalized to a value of 1.0, and the workload at all of the other nodes assigned pro-rata values between 0.0 and 1.0.

Processor	Batch				
	2	3	4	5	6
1	0.64	0.56	0.66	0.69	0.56
2	1.00	1.00	1.00	1.00	1.00
3	0.55	0.58	0.54	0.64	0.73
4	0.46	0.57	0.56	0.56	0.46
5	0.51	0.46	0.51	0.57	0.55
6	0.69	0.61	0.69	0.57	0.60
7	0.56	0.40	0.44	0.40	0.48
8	0.57	0.51	0.61	0.67	0.64
Imbalance	1.61	1.70	1.59	1.57	1.59

Table 1: Load imbalance for the pipelined retrieval system when terms are assigned to nodes using a single random assignment. Each batch reflects a simulated evaluation of the workload associated with 10,000 SYNQ queries, across $k = 8$ processors. The last row shows the ratio of the largest load to the average load, over that batch. The larger that value, the greater the imbalance.

The resulting normalized workload ratios for a single random assignment is shown in Table 1. For reasons that are explained shortly, results are not shown for batch one. The table clearly shows that, by luck, processor two has been assigned a combination of terms that results in it having the heaviest workload in all five of the batches for which results are given; on the other hand, processors four, five, and seven are half-idle.

The final row of Table 1 shows the factor by which the workload on the node with the heaviest load in each batch exceeds the average across all eight processors. The larger this value, the more likely it is that there will be nodes on the network that are being starved of useful computation. In turn, starvation results in lowered overall throughput; in an ideal situation, the peak:average workload ratio would be 1.0, and all nodes would share the workload equally. The imbalance of around 1.6 shown in Table 1 represents a starting point for the investigation reported in this paper, and a crystallization of the effect noted by Moffat et al. [2005].

5. REDUCING IMBALANCE

To smooth the workload across the processors, it is necessary to ensure that no one processor is responsible for an excessive number of high-workload terms. Two separate strategies suggest themselves: distributing the inverted lists such that the workload is evenly balanced; and duplicating inverted lists such that the same high-workload term is managed by multiple processors.

There are several possible approaches to an even-balance distribution of inverted lists. An obvious approximation is to base the term-to-processor assignment on the set of term frequencies f_t, on the assumption that the number of pointers in each inverted list roughly corresponds to the workload generated by that list when queries are being answered. This mechanism has the advantage of being query independent.

[1] See http://www.seg.rmit.edu.au.

Strategy	Batch					Avg
	2	3	4	5	6	
Random	1.45	1.44	1.46	1.50	1.48	1.47
Using f_t	1.43	1.20	1.23	1.40	1.42	1.34
Past L_t	1.14	1.26	1.23	1.19	1.17	1.20
Current L_t	1.00	1.00	1.00	1.00	1.00	1.00

Table 2: Estimated load imbalances with different term assignment strategies. Each batch reflects a simulated evaluation of the workload of 10,000 SYNQ queries, across $k = 8$ processors, as a ratio of the largest workload to the average workload, for that strategy. Imbalances for the "Random" row are the averages over 1,000 random assignments of terms to nodes. The last column shows the average imbalance over the five query batches.

Strategy	Batch					Avg
	2	3	4	5	6	
Duplicate 1	1.26	1.20	1.10	1.17	1.11	1.17
Duplicate 10	1.06	1.29	1.17	1.18	1.16	1.17
Duplicate 100	1.09	1.14	1.10	1.13	1.15	1.12
Duplicate 1000	1.08	1.09	1.07	1.19	1.09	1.10
Multi-replicate	1.05	1.12	1.09	1.16	1.12	1.11

Table 3: Estimated load imbalances with different amounts of index list replication. All assignment is via the fill smallest term assignment strategy based on the "Past L_t" approach (see Table 2). When multiple servers host a query term, a random choice is made between them by the receptionist. Results can be compared with the third row of Table 2.

Alternatively, the distribution can be based on the workload L_t, computed in arrears for some previous part of the query stream. In this "past L_t" method, workload assessment is computed at the end of each query batch, and assumed to generate a term assignment that is then used through the whole of the next query batch. (This is why no results are reported for query batch one in any of the tables – in this approach, batch one is used as the workload model for batch two, and so on.)

As an exploration of the potential gains available through redistribution, it also makes sense to consider a hypothetical "current L_t" system that knows (perhaps via an oracle of some sort) the workload distribution for each batch at the beginning of that batch.

Once per-term workload estimates have been prepared, there is also the question as to how to use them. The obvious approach is to assign terms to processors in a round-robin manner, to guarantee that the heaviest workload terms do not reside on the same machine. However, while such an approach puts equal numbers of terms on each machine, it does not result in balance. To see why, imagine that the heaviest term is temporarily held back, and all of the other terms assigned to nodes, starting from node two. If the round-robin approach yields a balanced set of workloads, this process should result in a balanced workload; but when the first term is then assigned to node one, significant imbalance must result. That is, the round-robin method does not in fact give balanced workloads.

Instead, we employed a *fill smallest* approach that considered each term in turn, from heaviest workload through to least, and assigned it to the machine that had (through until this moment) been assigned the least total workload. This approach results in nodes hosting different numbers of terms, but having very similar total workloads.

Table 2 shows simulated workload imbalance numbers for these strategies, compared to the starting point of random assignment (one example of which is shown in the last row of Table 1). As these results show, although the term frequency f_t is a significant component of the inverted list length B_t, which in turn is a factor in L_t, the query frequency count Q_t wields a much stronger influence. A static assignment of terms to nodes based on f_t alone provides performance only slightly better than a random assignment. At the other extreme, the "Current L_t" approach (naturally) attains a simulated workload imbalance average of 1.00.

A confounding issue is that it is possible for one term in the query stream to be so common that, even its inverted list were placed on a machine by itself, the workload would not be balanced. Splitting of lists across more than one machine, by separating them into parts, is not a satisfactory solution, as it increases network traffic. A more attractive option is to *replicate* lists, so that the receptionist is able to choose between alternative routings for a high-workload query, and so that the effort associated with processing a given list can be shared across multiple machines.

Replicating every list is unlikely to be satisfactory. Doing so would require doubling the storage at each machine, and would undermine the effectiveness of caching, as twice the volume of index data is active at each node. Moreover, most of the benefit of replication is likely to be observed by considering only the high-workload terms. We thus propose selectively replicating the inverted lists of a small number of high workload terms, potentially halving the peak workload associated with each copy of the list.

A first issue is the number of lists to replicate. Table 3 shows the effect of duplicating the lists of the highest-workload 1 to 1,000 terms. The final "multi-replicate" row shows a more complex arrangement in which the most frequent index term is placed on all eight processors; then another nine terms are placed on four of them; then another ninety terms are placed on two processors. As Table 3 shows, replication can indeed lead to better (simulated) workload balance. Replication is not guaranteed to improve performance, as can be seen by comparing the rows "Duplicate 1" and "Duplicate 10".

Another potential problem in Table 3 is that making a random choice (when choice is available) for the query routing may not be ideal. Rather, it makes sense to route each query to the alternative host that has the smallest workload, so that the system is adaptive to the balance of query terms in the current batch. Having the receptionist track assigned workload in this way represents a temporal equivalent of the "fill smallest" approach, and allows unexpected interactions between terms to be at least partially allowed for. That is, when a choice of routing is available, a better strategy is to send the query to the processor that has had the least volume of query work assigned to it so far.

Table 4 shows the effect of making this change. As can be seen, load-based query routing combined with replication can lead to almost perfect (simulated) load balance. Multi-replication appears to be unnecessary for this data and queries, but might be important if there were still serious load imbalances not addressed by mere replication. Nor does it seem likely that replicating every list would lead to further improvements in balance.

Replication does, of course, add to the total storage required by the index, especially since it is exactly the high-frequency terms that get duplicated. The next section reports on actual query throughput results for some of these options, including the additional storage cost incurred.

Strategy	Batch					Avg
	2	3	4	5	6	
Duplicate 1	1.26	1.20	1.09	1.17	1.11	1.17
Duplicate 10	1.03	1.16	1.07	1.07	1.09	1.08
Duplicate 100	1.01	1.02	1.01	1.03	1.02	1.02
Duplicate 1000	1.00	1.00	1.00	1.00	1.01	1.00
Multi-replicate	1.01	1.00	1.00	1.00	1.01	1.00

Table 4: Estimated load imbalances with different amounts of index list replication, using SYNQ queries. All assignment is via the smallest-first term assignment strategy based on the "Past L_t" approach (see Table 2). Load tracking (rather than random choice) is used to set the routing when queries contain replicated terms. Results can be directly compared with the matching rows of Table 3.

6. LIVE EXPERIMENTATION

A simulation is never better than the assumptions on which it is based, and sometimes much worse. We also implemented a document-distributed system, and the "Duplicate 100" distributed system, using as a starting point for both a carefully engineered version of the Zettair retrieval system.

The hardware used in all the experiments described in this section 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, supported 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. An important aspect of measurement in this area is to allow the machines to operate as efficiently as possible, and to this end all of the software used allows multi-threaded operation, with as many as 32 queries concurrently active at any given time.

We also paid particular attention to file placement on disk, having noted in previous work [Webber and Moffat, 2005] that variability in the way in which files were allocated could generate considerable volatility in experimental measurements. A dedicated "experimental" partition of the right size was maintained on the local disk of each of the nodes, and the relevant index file copied into it at the start of each experiment, so as to ensure that disk speed did not become a factor in the experiments.

To carry out each experiment, a query stream (one batch) was processed from beginning to end, after the previous batch was executed in non-timed mode as a warm-up. The top $r = 1{,}000$ scoring documents for each query were identified, and their TREC document identifiers output. All words and numbers in the source collection were indexed, and entered in a document-level index without ordinal word positions. The software system Zettair makes use of a byte-based compression regime that provides a compromise between compression effectiveness and decoding speed. The document-distributed index required 18.3 GB, including all vocabulary files; the term-distributed index occupied 16.1 GB; and the "Duplicate 100" term-distributed index required 16.7 GB.

Table 5 shows normalized query throughputs, measured in units of terabyte queries per machine second. We use this unit of *normalized throughput* as our yardstick (rather than *unnormalized throughput*, measured in queries per second), because it takes into account the two key factors that affect query rates in a scaling sense, namely, the size of the collection, and the number of machines applied. For example, if a cluster of $k = 8$ machines working with the GOV2 collection processes a batch of 10,000 queries in 115 seconds (a typical run time for our experiments), then the unnormalized throughput rate is 87.0 queries per second, and the normalized throughput rate (taking into account collection size and number of processors in use) is 4.52 terabyte queries per machine second. If we had wished to be even more precise, we could have replaced "machines" in this computation by "total processing gigahertz"; however, all the machines in our cluster have the same speed, and we use the simpler computation.

Strategy	Batch					Avg
	2	3	4	5	6	
Hashed	4.04	4.05	4.29	4.34	4.10	4.16
Duplicate 100	5.09	5.00	5.28	5.34	5.41	5.22
Doc-distributed	5.36	5.50	6.02	6.07	5.82	5.75

Table 5: Measured query throughput rates on 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, where each batch consists of 10,000 SYNQ queries executed against the GOV2 collection to identify the top $r = 1{,}000$ matching documents, and where the numbers reported are in units of terabyte queries per machine second. As many as 32 simultaneous query threads were permitted. The last column shows the average throughput over the five query batches.

Strategy	Batch					Avg
	2	3	4	5	6	
Hashed	4.66	4.60	4.69	4.59	4.36	4.58
Duplicate 100	6.13	5.84	5.83	5.91	6.00	5.95
Doc-distributed	7.86	7.87	7.90	7.89	7.81	7.87

Table 6: Measured sum of busy loads on processors, not including the receptionist, for SYNQ queries.

The results in Table 5 show that selective replication based on past workload, combined with load-based query routing, is a clear improvement on random unreplicated term assignments. However, despite the 30% gain in performance, pipelining still falls short of the consistently high throughput rates achieved by document-distributed indexing.

To try and understand why, Table 6 shows the sum, across the eight processors (not including the receptionist), of the measured busy time while each query batch is being processed, where "busy" includes all non-idle activities. A value of 8.0 in this table would indicate that the entire system was completely saturated with computation throughout the processing, an outcome that would only be attainable if all nodes were busy all of the time. Values below 4.0 indicate that more than half of the available processing resource is wasted, and that nodes are being starved of work. As can be seen, in document distribution the machines come very close to being fully utilized, while for "Duplicate 100" pipelining the utilization is just under 75%.

Table 7 shows the measured workload imbalance for the three distributed retrieval schemes that were tested. These results show

Strategy	Batch					Avg
	2	3	4	5	6	
Hashed	1.63	1.67	1.59	1.67	1.75	1.66
Duplicate 100	1.06	1.06	1.07	1.05	1.10	1.07
Doc-distributed	1.01	1.01	1.01	1.00	1.02	1.01

Table 7: Measured workload imbalance across the eight processors, for SYNQ queries.

Strategy	Batch					Avg
	2	3	4	5	6	
Hashed	1.82	1.82	1.86	1.84	1.83	1.83
Duplicate 100	2.21	2.14	2.25	2.17	2.20	2.19
Doc-distributed	2.21	2.25	2.24	2.31	2.27	2.26

Table 8: Measured query throughput rates for GOVQ queries. Other details are identical to Table 5.

that our earlier simulations – which were used to help determine which systems were worth full-scale implementation and testing – were indicative but not completely accurate. Unreplicated pipelining was worse than predicted, while replication achieved slightly better performance than we anticipated. These experiments illustrate that simulation is no substitute for live measurement, as the impact of factors such as caching is extremely difficult to model.

Another possible limitation in our experiments was the use of artificial queries, despite the care with which these queries were constructed [Webber and Moffat, 2005]. We also had access to a collection GOVQ of queries provided to us by Microsoft Search; these are queries extracted from their search log for which one of the top three ranked document was in the `.gov` domain. This log is also somewhat artificial (due to possible selection bias); nonetheless it is a set of real queries that can be used for interrogating the GOV2 crawl. The queries in the log were stopped to remove all occurrences of the commonest six terms in the collection, and a further stoplist of 310 words was used to stop all queries except for those that would be reduced below three words.

Table 8 is identical in structure to Table 5, but uses these GOVQ queries. The results are broadly consistent. Note that, due to the different distribution of query terms and frequencies, throughput is much lower than for the SYNQ queries.

The reason why the replicated pipelined method is unable to better the document-distributed mechanism, even though the CPUs are less busy, comes back down to load balancing, but now on a burst level rather than a batch level, where a "burst" is defined to be a short sequence of queries, perhaps a second's worth. Over short spans of time there are micro-imbalances in workload caused by random chance in the query stream, with the query routings for each burst often colliding at one or the other of the nodes handling them. Figure 2 plots the average busy load over bursts of 100 queries at a time for a sequence of 2,000 queries, and the busy load for one of the nodes. The average busy load is relatively steady, but the single machine uses a variable amount of CPU time during each of the query bursts.

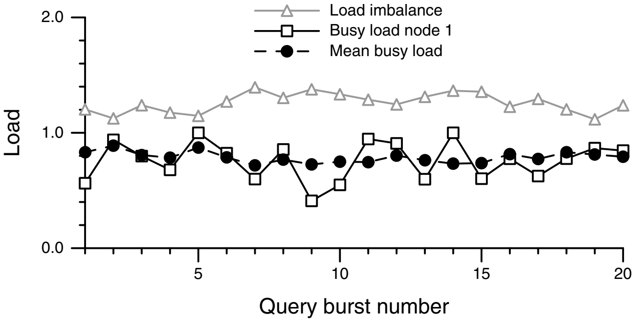

Figure 2: Busy load measured locally for twenty query "bursts" each containing 100 queries, plotted for a single node and compared to the average busy load across all 8 nodes. The upper line shows the load imbalance in that query burst.

To allow for this volatility, we also tested a "decaying workload" variant of the system, in which the estimates of total node workload that were used as the basis for the routing decisions were slowly eroded over time, so that the receptionist gives recent information more weight, thereby perhaps eliminating localized hotspots. However, we were unable to substantially improve on the throughput rates already reported. It would appear that workload imbalance at the micro level remains an issue in the pipelined approach, even after the workload imbalance at the macro level has been addressed.

7. SCALABILITY

One issue that is potentially very hard to deal with is that of scalability. Linear growth in resource consumption as problem size increases is a desirable attribute of any algorithm, but is something that requires delicate argument when both data sizes and processing power are simultaneously being increased. For example, it may not be appropriate to show scalability by taking a fixed (even if large) amount of data and showing that the time taken to solve a problem is proportional to $1/k$ when k processors are used – the speed up might, for example, have been achieved solely as a function of there being more main memory (across the pool of machines) in which fixed-size data structures can be accommodated.

More useful is to grow data volume and processor numbers at the same rate, and ask whether performance can be maintained at the previous levels. This approach is in contrast to the results reported in Section 6, where the maximum amount of available data is applied across the whole set of available machines. Table 10 shows another view of distributed processing using the two methods (document-distributed and pipelining using the "Duplicate 100" approach). To construct the table, fractional collections were indexed on subsets of the processors, and query throughput rates measured. (Note that, to avoid excessive index rebuilding, we made one simplification to the pipelined system, and instead of building the index for each batch of 10,000 queries based on the "Past L_t" approach, we built a single index based on the workloads measured in the first batch of 10,000 queries, and then used that term assignment for all five subsequent batches.)

Down the lead diagonal in each part of the table, data volume grows in proportion to the number of processors, and in each of those four experiments, each machine has a constant-sized set of index information to manage. Normalized throughput is roughly constant, indicating that the communications overhead with either method is small.

Conversely, across the bottom row of each part of the table, a fixed number of processors is used to index a growing total amount of data, so that the amount handled on each machine also grows. In these rows normalized throughput increases, showing that on a single machine there are economies of scale to be obtained as data volumes increase. For very small collections (when each node has $426/64 = 6.7$ GB), the pipelined system outperforms the document-distributed one, but at a relatively low normalized throughput rate. If the number of simultaneous query threads is increased, pipelining also gains a relative advantage. For example, when 64 threads are active the throughput of the pipelined system in the "$k = 8$, 1/1" entry of Table 10 rises to 6.48 terabyte queries per machine second, compared with 6.42 for the document-distributed system.

Moving in the other direction, it is then interesting to speculate as to which factors become important as data volumes and processor numbers increase beyond the levels at which we are able to experiment. We do this via a two-stage "thought experiment", in which we first suppose that 10 times as much data (4.3 TB rather

Processing mode	When data volume increases 10-fold	When data volume increases 100-fold, and k increases 10-fold
Doc-distributed	Each node has 10 times as many documents, and node response takes 10 times longer. Unnormalized throughput drops by a factor of 10; normalized throughput is unchanged. Receptionist handles $1/10$ as many queries, and spends the same time on each one. Network traffic also drops by a factor of 10. Summary: query response time = $\times 10$, normalized throughput = $\times 1$.	Each node has 10 times as many documents, and node response takes 10 times longer. Unnormalized throughput drops by a factor of 10; normalized throughput is unchanged. Receptionist handles $1/10$ as many queries, but spends 10 times longer on each one. Network traffic is unchanged. Summary: query response = $\times 10$, normalized throughput = $\times 1$.
Pipelined	Each node has 10 times as much index data for the same number of terms, so node processing times increase by a factor of 10. Unnormalized throughput drops by a factor of 10; normalized throughput is unchanged. Receptionist handles $1/10$ as many queries, and spends the same time on each one. If query package size grow linearly with collection size, network traffic is unchanged. Summary: query response = $\times 10$, normalized throughput = $\times 1$.	Each node has 100 times as much index data for $1/10$ as many terms, so node processing times increase by a factor of 100. Unnormalized throughput drops by a factor of 10; normalized throughput is unchanged. Receptionist handles $1/10$ as many queries, and spends the same time on each one. If query packages grow linearly with collection size, network traffic increases by a factor of 10. Summary: query response = $\times 100$, normalized throughput = $\times 1$.

Table 9: Scalability options: in the first column, data volume alone is assumed to increase by a factor of 10; in the second column an additional 10-fold growth in data volume (100-fold in total) is coupled with a simultaneous 10-fold increase in the number of processing nodes. Note that query response time estimates assume that the system is not operating at peak throughput.

Strategy	Total collection size			
	1/8	1/4	1/2	1/1
Document-distributed				
$k = 1$	5.89	–	–	–
$k = 2$	–	5.81	–	–
$k = 4$	–	–	5.86	–
$k = 8$	3.19	4.35	5.22	5.75
Duplicate 100				
$k = 1$	5.89	–	–	–
$k = 2$	–	5.96	–	–
$k = 4$	–	–	5.69	–
$k = 8$	4.07	4.50	4.91	5.24

Table 10: Query throughput on fractional collections and processor subsets, with 32 query threads active. Each value is the average (over batches 2–6) of the normalized throughput, measured in units of terabyte queries per machine second. Term assignments in the pipelined method were based on batch one alone, rather than the immediately prior batch.

than 426 GB) is to be indexed on the same $k = 8$ machines as used in our experiments; and then, in a second stage of growth, that another factor-of-10 increase in data, to 43 TB, is accompanied by a corresponding growth in processors, to make $k = 80$. We suppose throughout this discussion that the query mix is unchanged, and that users continue to expect the system to respond with a list of the r most highly ranked documents. In the sense of Table 10, the two steps are represented as a further move to the right starting at the bottom element in the last row, and then, in the second step, a diagonal move down from that new point.

The question is this – what happens to overall query throughput rates? Are there aspects of either document-distributed or pipelined systems that become problematic? Table 9 summarizes our beliefs in this regard, and highlights a key issue with the pipelined approach – because queries are processed sequentially, there is no gain in response time to individual queries from parallelism. That is, when the data size grows by a factor of 10 or 100, so too must the response time to queries, regardless of how many processors are applied. This behavior is in contrast to the performance of the document-distributed system, where adding more processors to a system has the benefit of reducing individual query response times.

There is one important caveat to be added to this discussion, and that is to note that response times can only be low in either system if they are operating at less than their peak throughput rates. When any system is heavily loaded, with a queue of pending queries, the latency between initiation and completion of each query is directly proportional to the number of query threads that are simultaneously active, and inversely proportional to the unnormalized throughput. That is, per-query response time is an unhelpful concept in a maximally-loaded system taking inputs one-by-one from a queue of waiting queries.

8. PREVIOUS WORK

There is a substantial literature on distribution methods. Most of these papers concern document distribution. Harman et al. [1991] describe a document-distributed system that was successfully deployed in practice. Cahoon and McKinley [1996] and Cahoon et al. [2000] found that increasing the number of nodes used to manage a fixed-size collection can improve response, with diminishing returns; however, increasing the number of nodes without increasing the collection size leads to results that cannot be meaningfully interpreted, and that is the approach we have also taken here. A particular issue with much of the previous experimentation is that, due to the many non-linear properties of such systems, the behavior as collection size is increased is unknown.

Probably the best-known document-distributed system is Google [Barroso et al., 2003], in which the a cluster of nodes maintain a document-distributed index and other nodes store information such as the documents themselves. The document distribution provides fast response time; replication of clusters provides high throughput.

Some distribution methods can be categorized as hybrids. Pipelining [Moffat et al., 2005] has already been described. Another hybrid method is that of Xi et al. [2002a,b], where each inverted list is broken into k fixed-size chunks and one chunk is held on each node. A difficulty with this approach is that it has disadvantages compared to both document distribution (where each node completely indexes a sub-collection, so processing can be node-oriented) and term distribution (where the number of disk accesses is minimized). Whether there are advantages is unclear. Similar methods based on architectures such as connection machines [Cringean et al., 1990, Stanfill, 1990] have the same issues.

Much of the literature on term-distributed methods consists of papers comparing the term- and document-distributed approaches. Unfortunately, this literature is inconsistent. Using artificial data, Jeong and Omiecinski [1995] found in favor of document distribution. On real data, but only 50 queries – insufficient to show caching effects – MacFarlane et al. [2000] found the same result. However, Tomasic and García-Molina [1996] found that replication was needed for improvement in throughput. Contradicting all of these results, both Ribeiro-Neto and Barbosa [1998] and Badue et al. [2001] found that term distribution is superior.

However, much of the previous work is open to question. Artificial or small sets of data or queries are not likely to be predictive of real-world behavior, and simulations designed to estimate time must deal with a great many complex variables – including caching, relativities of CPU speed, network bandwidth, network delay, disk properties, term skew, and query skew – if they are to be realistic. For example, such issues undermine the results of Cacheda et al. [2004], who use simulation to compare distribution with replication, but neglect caching effects.

Hawking [1997], Melnik et al. [2001], and Lu and McKinley [2003] have also contributed to the literature on distributed indexing and querying.

9. CONCLUSION

We have explored ways in which the load issues associated with the pipelined distributed evaluation approach can be addressed. The final mechanism involves a blend of advance workload estimation, judicious list replication, and adaptive workload monitoring. These techniques increase the throughput of term-distributed indexing by 30%. Nevertheless, local fluctuations in workload mean that each node in the network is less than 100% busy, and while the final throughput rates attained in our experiments remain tantalizingly close to the rates achieved by an equivalent document-distributed computation, we did not succeed in beating document distribution, despite the heavier CPU consumption of the latter. An important conclusion of our investigation to date is thus that document-partitioning retains its leading position as the method against which others must be judged.

Acknowledgment. This work was supported by the Australian Research Council. We also thank Microsoft Search for the GOVQ log.

References

C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In G. Navarro, editor, *Proc. String Processing and Information Retrieval Symp.*, pages 10–20, Laguna de San Rafael, Chile, November 2001. IEEE Computer Society.

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

F. Cacheda, V. Plachouras, and I Ounis. Performance analysis of distributed architectures to index one terabyte of text. In S. McDonald and J. Tait, editors, *Proc. ECIR European Conf. on IR Research*, pages 395–408, Sunderland, UK, April 2004. Springer-Verlag. LNCS volume 2997.

B. Cahoon and K. S. McKinley. Performance evaluation of a distributed architecture for information retrieval. In H.-P. Frei, D. Harman, P. Schäuble, and R. Wilkinson, editors, *Proc. Nineteenth Annual International ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 110–118, Zurich, Switzerland, August 1996. ACM Press, New York.

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

J. K. Cringean, R. England, G. A. Manson, and P. Willett. Parallel text searching in serial files using a processor farm. In J. L. Vidick, editor, *Proc. Thirteenth Annual International ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 429–453, Brussels, Belgium, September 1990. ACM Press, New York.

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

D. Hawking. Scalable text retrieval for large digital libraries. In C. Thanos, editor, *Proc. European Conf. on Research and Advanced Technology for Digital Libraries*, pages 127–145, Pisa, Italy, September 1997. Springer-Verlag. LNCS volume 1324.

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

N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In *Proc. Sixth Int. Conf. on Web Informations Systems*, pages 470–477, New York, November 2005. LNCS 3806, Springer.

Z. Lu and K. S. McKinley. Partial collection replication for information retrieval. *Kluwer International Journal of Information Retrieval*, 6(2): 159–198, 2003.

A. MacFarlane, J. A. McCann, and S. E. Robertson. Parallel search using partitioned inverted files. In P. de la Fuente, editor, *Proc. String Processing and Information Retrieval Symp.*, pages 209–220, A Coruña, Spain, September 2000. IEEE Computer Society Press, Los Alamitos, California.

S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. *ACM Transactions on Information Systems*, 19(3):217–241, 2001.

A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. September 2005. Submitted.

B. Ribeiro-Neto and R. Barbosa. Query performance for tightly coupled distributed digital libraries. In I. Witten, R. Akscyn, and F. M. Shipman III, editors, *Proc. ACM Digital Libraries*, pages 182–190, Pittsburgh, Pennslyvania, June 1998. ACM Press, New York.

C. Stanfill. Partitioned posting files: a parallel inverted file structure for information retrieval. In J. L. Vidick, editor, *Proc. Thirteenth Annual International ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 413–428, Brussels, Belgium, September 1990. ACM Press, New York.

A. Tomasic and H. García-Molina. Performance issues in distributed shared-nothing information-retrieval systems. *Information Processing & Management*, 32(6):647–665, 1996.

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

W. Xi, O. Sornil, and E. A. Fox. Hybrid partition inverted files for large-scale digital libraries. In *Proc. Digital Library: IT Opportunities and Challenges in the New Millennium*, pages 404–418, Beijing, China, July 2002a. Beijing Library Press.

W. Xi, O. Sornil, M. Luo, and E. A. Fox. Hybrid partition inverted files: Experimental validation. In M. Agosti and C. Thanos, editors, *Proc. European Conf. on Research and Advanced Technology for Digital Libraries*, pages 422–431, Rome, Italy, September 2002b. Springer-Verlag. LNCS volume 2458.