Design and Evaluation of a Pipelined Distributed Information Retrieval Architecture

William Edward Webber
Department of Computer Science and Software Engineering
University of Melbourne

Submitted in total fulfilment of the requirements
of the degree of Master of Engineering

Produced on acid-free paper

June 2007

Abstract

Web-scale search engines deal with a volume of data and queries that forces them to make use of an *index* partitioned across many machines. Two main methods of partitioning an index for distributed processing have been described in the literature. In *document partitioning*, each processor node holds the information for a subset of documents, while in *term partitioning*, each node holds the information for a subset of terms. Additionally, a novel architecture, *pipelining*, has been proposed, offering to combine the best features of both architectures.

This thesis develops a careful methodology for the experimental comparison of distributed information retrieval architectures, addressing questions such as experiment scalability and query set generation. Novel methods are proposed for accumulator pruning, and for compression of accumulators for shipping between nodes in the pipelined architecture.

A meticulous experimental assessment of the three distributed architectures is then undertaken. The results demonstrate that term distribution suffers a severe processing bottleneck. Pipelining resolves term distribution's processing bottleneck, while maintaining its superior I/O characteristics. However, pipelining suffers from serious load imbalance between the nodes, fails to fully utilise the cluster's processing capacity, and scales poorly. Document distribution, in contrast, distributes workload evenly and scales well.

Load balancing through the intelligent assignment of terms to partitions is explored, but fails to fully resolve the imbalance of the pipelined architecture. Instead, the partial replication of high-workload terms is proposed, coupled with the intelligent routing of queries. These techniques resolve pipelining's load imbalance, allowing it to marginally outperform document distribution.

The partially-replicated pipelined architecture is shown to benefit from system scale. It also significantly outperforms document distribution in a memory-limited environment, suggesting that it would also outperform for larger collections relative to cluster size. However, unlike document distribution, pipelining's average response time at low to moderate loads does not scale. The system implementor must therefore weigh the relative benefits of the two architectures.

Declaration

This is to certify that

(i) this thesis contains only my original work towards the degree of Master of Engineering except where indicated in the Preface;

(ii) due acknowledgement has been made in the text to all other material used; and,

(iii) this thesis is of approximately 30,000 words in length, exclusive of tables, maps, bibliographies, and appendices.

William Webber
Department of Computer Science and Software Engineering
University of Melbourne
June 22, 2007

Acknowledgments

Thanks go first of all to my supervisor, Professor Alistair Moffat, $\pi\o\lambda\upsilon\mu\eta\tau\iota\varsigma$, a few minutes of whose advice was frequently of more benefit than many hours of my own efforts. Thanks go also to my second advisor, Professor Justin Zobel of RMIT University, without whose support I would never have started this project, and without whose encouragement I would scarcely have completed it.

I have been fortunate to have had as my senior colleague and mentor Dr Vo Ngoc Anh. His advice and support through the period of my candidature, and his scrupulous reading and correction of my work in this thesis, has made him nothing less than a third supervisor to me.

It has been my good fortune to enjoy the company, support, and kindness of numerous colleagues in the Department: Amy Unruh, Ce Dong, Elsa Loekito, Laurence Park. Mike Ciavarella, Robert Shelton, Shane Culpepper, Tony Wirth, Yi Li, and YuYe Zhang. Particular thanks must go to Nicholas Lester, the lead developer of the Zettair system, which has formed the basis of the software used throughout the thesis.

Many friends have greatly helped me over the past eighteen months. My appreciation goes to Pauline Chou, Joanne Zhu, and Yvonne Ren, for their support and encouragement. Thanks also go to my family, in particular my mother Jeananne Webber.

I would like to acknowledge the Australian Research Council, under whose funding I was employed during the period of my candidature, and also to the Department of Computer Science and Software Engineering for travel support, and for the facilities made available to me.

This thesis would not have been possible without the support of all of these people; but the remaining deficiencies are my own.

Preface

Publications arising from this thesis

The work on appropriate and inappropriate query sets and the generation of synthetic queries in Section 4.1, on scalability of experimental design in Section 4.4, and on controlling disk access variability in Section 4.5, was presented at the 10th Australasian Document Computing Symposium [Webber and Moffat, 2005].

The dynamic query pruning technique described in Section 5.2 was presented at the 6th International Conference on Web Information Systems [Lester et al., 2005]. This work was done jointly with Nick Lester and Justin Zobel of RMIT University, and is also incorporated in Nick Lester's recently submitted PhD Thesis, "Efficient Index Maintenance for Text Databases" [Lester, 2006].

The experimental assessment of the document-distributed, term-distributed, and pipelined architectures presented in Chapter 6 has been published in the journal Information Retrieval [Moffat et al., 2006b]. This thesis uses a later version of the software and some different experimental settings, and so the precise timings differ from those presented in the journal paper, although the relativities are the same. This work was done jointly with Justin Zobel of RMIT University and Ricardo Baeza-Yates of the University of Chile. Note that the notion of pipelining as a distributed evaluation technique was formulated prior to my involvement in the project, and my role starts with the experimental evaluation.

Chapter 7's investigation of load balancing in a pipelined system is an extension of work presented at the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval [Moffat et al., 2006a]. This work was done jointly with Justin Zobel of RMIT University.

Document preparation, tools and data

This document was prepared with TEX, LATEX and BIBTEX, using the `setspace`, `multirow`, `bigbrace`, `algorithmic`, `algorithm`, `natbib`, `epsf`, `url`, `dcolumn`, and `texnames` packages. Figures were drawn using `xfig`, and graphs produced with `jgraph`.

Experiments were run under the Debian GNU/Linux operation system. The Zettair information retrieval engine developed at RMIT University was used as the basis for system implementation. Programs were compiled using the GNU C compiler. The Python programming language was used extensively for managing experiments and collating results.

The query sets for the efficiency runs are based on a log provided by Excite, Inc. The other raw data for these experiments was produced by the United States National Institute of Standards and Technology via their TREC program.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Knowledge is of two kinds.
We know a subject ourselves,
or we know where we can find information upon it.
Samuel Johnson, from Boswell, *Life of Johnson*, 1791.

The World Wide Web has become the nexus of the information age, and web search engines are a ubiquitous part of modern life. Users expect that a web search engine will be comprehensive, and that it will provide results almost instantaneously. At the same time, the web already contains billions of documents, and is growing faster than the processing capacity of hardware. The mass of data to index and the volume of queries to support force the deployer of a web-scale search engine to distribute processing across a large number of machines. So immense are the processing requirements of such a system, that even a small improvement in the efficiency of query evaluation can result in large savings in hardware and operational costs.

Fortunately, ranked query evaluation (the information retrieval technique implemented by modern search engines) is a readily distributable task. Updates to the search engine's index can be batched and made off-line. As a result, search engines operate on an index that is essentially read-only, and so can easily be replicated. In addition, query processing is a largely decomposable task. The index can be broken up into pieces, and processing performed on each piece in a substantially independent way.

Two main methods of partitioning an inverted index (the form of index employed by modern search engines) have been described in the research literature. The most straightforward and perhaps most obvious one is to partition the index into subsets of documents, where each partition is a largely autonomous index of one part of the document collection. *Document partitioning* offers the cleanest decomposition of the query evaluation task. Queries can be evaluated upon each partition of the index with a very high degree of independence from the other partitions. However, some researchers have suggested that an alternative method of partitioning the index offers superior efficiency. In *term partitioning*, the index is broken up into subsets of terms, with each partition handling a separate portion of the index's vocabulary. The decomposition is less clean than in document partitioning, as query evaluation requires the cooperation of all subindexes holding term information for the terms of a given query, for instance by shipping this information to a central node for processing. The compensation is that term information is not fragmented across multiple partitions, promising better disk access characteristics.

A number of studies have compared the document-wise and term-wise methods of partitioning an inverted index, both in simulation and (less commonly) in implementation. However, the results of these studies have been inconsistent. The superior data contiguity of term partitioning is attractive, but is the central processing of the index information a scalable solution, or an effective use of the cluster's processing capacity? And is there a way of combining term partitioning's desirable data arrangement characteristics with document partitioning's ready decomposition of the processing task? These are the questions that this thesis sets out to answer.

Thesis structure

The thesis is organised as follows. A brief overview of the field of information retrieval is provided in Chapter 2. The problem domain that research into information retrieval addresses is sketched, and the most essential items of terminology are introduced. Information retrieval uses query evaluation models that both provide a sound theoretical basis for the discipline, and also inform pragmatic implementation decisions; some of the most popular models are described in Chapter 2. Since the focus of the thesis is on the efficient implementation of distributed processing, a firm understanding of index construction and query processing mechanisms is an essential background; such an understanding is provided in Chapter 2. Finally, the standard datasets and tools developed by the research community for evaluating the effectiveness of different information retrieval techniques, particularly centering around the TREC effort, are introduced.

Chapter 3 describes the current state of the art in distributed information retrieval. The motivation for, and aims of, the sub-discipline are outlined. The two main existing distribution methods are described, namely document-wise and term-wise distribution. A novel architecture, called pipelining, is then introduced. Next, previous research in the area of distributed information retrieval is surveyed, and finally, the research contributions of the thesis are outlined. Chapter 3 concludes the introductory portion of the thesis.

Performing reliable experiments on distributed information retrieval systems requires some care. A thorough methodology for distributed information retrieval experiments is introduced in Chapter 4. A variety of issues are addressed, including query set selection or generation, query processing parallelism, scalability, and disk access variability.

The pipelined architecture ships the query processing state between nodes in the cluster. The processing state consists primarily of partially-evaluated document scores or *accumulators*, and since the number of them can potentially be very large, their shipping could become a major bottleneck for a pipelined system. Chapter 5 describes compression techniques for reducing the size of the shipped accumulators, and proposes a novel dynamic pruning regime for limiting the total number of accumulators created in the course of evaluating each query.

The experimental methodology developed in Chapter 4 is deployed in Chapter 6, in a careful evaluation of the document-distributed, term-distributed, and pipelined architectures. The experiments test the different architectures with varying collection and cluster sizes. A post-mortem on the (initally less than encouraging) results for the pipelined architecture determines that pipelining has a serious problem with processing load imbalance between the nodes of the cluster.

Possible solutions to pipelining's poor balancing of load are examined in Chapter 7. Load balancing through explicit term assignment is first examined, and found to be insufficient. Instead, partial term replication and intelligent query routing are proposed, and demonstrated to largely solve pipelining's load balance problems.

Chapter 8 is the last of the experimental chapters. It tests the scalability of the partially-replicated and intelligently-routed pipelined architecture. Particular attention is paid to operation in a memory-limited environment, and the scalability of average response time in the pipelined and document-distributed architectures.

Finally, Chapter 9 summarises the findings of the thesis, and proposes directions for future research.

Chapter 2

Information retrieval

This chapter provides a brief introduction to the theory and implementation of ranked query evaluation. The first section gives an overview of the field of information retrieval and the place of text retrieval and ranked query evaluation within it. In the second section, the use of similarity metrics to assess the relevance of documents to queries is introduced, and one commonly-used similarity metric, the cosine measure, is described. Section 2.3 then examines the construction of indexes to support ranked query evaluation, and the processing regimes that operate upon these indexes. And finally, metrics for assessing the effectiveness of a retrieval system are described in the fourth section. These sections provide the necessary background for understanding the examination of distributed information retrieval that will follow in Chapter 3.

2.1 Background

The field of information retrieval is broadly concerned with using information sources or data to meet people's information needs. The user expresses their information need in the form of a *query*; the information retrieval system *evaluates* the query with reference to its information pool or *collection*; and on the basis of the evaluation, the information retrieval system provides the user with a set of data that answers the query. The process may be repeated in an iterative and exploratory way.

The collection of data that an information retrieval system manages may be structured, such as in a relational database system, in which case the queries can manipulate the structure, using, for example, SQL. Alternatively, the data may be semi-structured or unstructured, as with web pages and plain text documents. In this case, the user's query generally takes the form of a series of words, or *terms*, that describe what they are searching for. Such a query may contain

Algorithm 1 : core query evaluation algorithm

```
Input: a collection of documents $\mathcal{D}$, and a query $q$
assign $\mathcal{A} \leftarrow \{\}$
for each document $d \in \mathcal{D}$ do
    assign $s_d \leftarrow sim(q, d)$
    assign $\mathcal{A} \leftarrow \mathcal{A} \cup (s_d, d)$
sort $\mathcal{A}$ by decreasing $s_d$
Output: top $r$ documents of $\mathcal{A}$
```

Boolean operators and phrases, or may be a *free text query*, made up simply of a list of terms.

Free-text querying has become particularly popular and important with the advent of the World Wide Web as an enormous, semi-structured data source, and of search engines as aids to users attempting to locate information within it. There has been recent interest in attempting to use the Web to directly answer users' questions; for instance, by answering the question "what is the capital of Spain?" with the answer "Madrid". However, traditionally, free-text information retrieval engines have sought to answer user queries by presenting a document or list of documents considered likely to be *relevant* to the user's query. Typically, degree or likelihood of relevance is estimated via a scoring function (although not necessarily reported), and the user is presented with the answer documents ranked in decreasing estimated relevance. See Witten et al. [1999], and Baeza-Yates and Ribeiro-Neto [1999], for an overview of the field of information retrieval.

The above discussion can be summarised as follows:

Definition 1 *A full-text search engine takes a user's query q, consisting of discrete query terms $\{t_1, \cdots, t_{|q|}\}$; evaluates it against a document collection $\mathcal{D}$, consisting of documents $\{d_1, \cdots, d_N\}$; and answers it with a ranked list of documents $\{a_1, \cdots, a_r\}, a_i \in \mathcal{D}$, ordered in decreasing estimated relevance to the query q.*

The rest of the thesis assumes a search engine performing as described in Definition 1.

2.2 Models and metrics

A simple processing model for a search engine as described in Definition 1 is provided in Algorithm 1. The degree of relevance of each document to the query q is estimated by the *similarity metric*, *sim*. A similarity metric takes a $\langle q, d\rangle$ pair as input and returns a similarity score as output. Then the documents are ordered by their similarity score, and the top r documents are returned to the user in this order.

A large number of different similarity metrics are described in the literature. Zobel and Moffat [1998] identify eight different components of a similarity metric, combining to create in the order of 100,000 mathematically distinct measures—and that is not including the recently-developed language model formulations. A common factor to most metrics is that they derive the similarity of a document to a query from the relative or absolute frequency of query terms in the document and in the collection. Each query term t occurring in document d is assumed to have a contribution $w_{d,t}$ to that document's similarity score. Most metrics derive $w_{d,t}$ using a "*tf·idf*" formulation, which states that the term's contribution $w_{d,t}$ is positively correlated with the term's frequency within the document (*tf*), and negatively correlated with the number of documents it occurs in (*idf* or *inverse document frequency*) [Salton and McGill, 1983].

An early and popular model for evaluating document-query relevance that encodes the *tf·idf* principle is the vector space model [Salton et al., 1975]. In this method, a document is modelled as a vector in multi-dimensional space, where each distinct term corresponds to a dimension, and the document's coordinate in that dimension is defined by the respective $w_{d,t}$ value. The query is modelled in the same way. The similarity score between a document and a query is then calculated as the closeness of their respective vectors. The closer the two vectors are, the more similar the document and the query, and hence the more relevant the document is estimated to be to the query. Conventionally, the vector similarity is calculated as the cosine of the angle between the vectors, which is the same as the length-normalised inner product of the vectors:

$$\begin{aligned} sim(q,d) = cosine(\vec{q},\vec{d}) &= \frac{\vec{q}\cdot\vec{d}}{|\vec{q}||\vec{d}|} && (2.1) \\ &= \frac{1}{W_q W_d}\sum_{t=1}^{n} w_{q,t}\cdot w_{d,t} && (2.2) \end{aligned}$$

where

$$W_d = \sqrt{\sum_{t=1}^{n} {w^2}_{d,t}}$$

and similarly for W_q, and where $\vec{q}$ and $\vec{d}$ are the vector representations of q and d, respectively.

The class of metrics derived from Equation 2.2—with some suitable formulation for $w_{d,t}$ and $w_{q,t}$—are collectively termed the "cosine measure". More recently, interest has turned to metrics based upon probabilistic models, such as Okapi BM25 [Robertson et al., 1994], and upon language models. Indeed, the relevance metric used in the system used for the experiments reported in this thesis is based upon a unigram language model with Bayesian smoothing using Dirichlet priors [Zhai and Lafferty, 2004]. However, since choice of metric does not greatly affect the distributed algorithms proposed in this thesis, and since the cosine measure is the simplest and longest established, it will be used here for exposition.

To calculate the document and query vectors, formulations for $w_{d,t}$ and $w_{q,t}$ must be chosen. A number of choices are possible. The following formulations, given by D'Souza et al. [2000], are typical:

$$\begin{aligned} w_{d,t} &= \log(f_{d,t} + 1) \qquad (2.3) \\ w_{q,t} &= \log(f_{q,t} + 1) \cdot \log\left(\frac{N}{f_t} + 1\right) \qquad (2.4) \end{aligned}$$

where $f_{d,t}$ is the number of times term t occurs in document d, $f_{q,t}$ is the number of times term t occurs in the query q, N is the total number of documents in the collection, and f_t is the number of documents that term t occurs in. Note that in the above, *tf* is encoded in both $w_{d,t}$ and $w_{q,t}$, but *idf* is only encoded in the latter. Substituting these formulas into Equation 2.1, we get:

$$sim(q, d) = \frac{1}{W_d} \sum_{t \in q \cap d} (\log f_{d,t} + 1) \cdot (\log f_{q,t} + 1) \cdot \log\left(\frac{N}{f_t} + 1\right) \qquad (2.5)$$

where W_q has been dropped, as it is the same for each document for a given query, and so does not affect ranking.

Current web search engines also use a range of other features to rank documents, including evidence of document quality such as PageRank [Brin and Page, 1998]. Such techniques have not been implemented in the experimental system used here; however, a discussion of how they might be implemented in a distributed environment can be found in Section 5.4.

2.3 Indexes and processing

In order to enable the efficient processing of user queries, an index of the collection is built. The most common form of index for full-text query evaluation is the *inverted index* [Fox et al., 1992, Zobel et al., 1998, Witten et al., 1999]. For *tf·idf*-based relevance metrics, such as the cosine measure given in Equation 2.5, the information that must be collected for each term is its frequency throughout the collection; which documents the term occurs in; and how often it occurs in each document. Each document occurrence and frequency pair is referred to as a *pointer* or *posting*, and the list of them is called an *inverted list*. Within inverted lists, documents are referred to by number, and a separate data structure known as a *docmap* is used to translate from document number to the full document name (for instance, the URL of the document), and also to provide document statistics if required for query evaluation (for instance, the document length for length-normalised similarity metrics). Finally, there needs to be a way to quickly access information for a given term, which is generally provided by a separate *vocabulary*. A common choice for implementing the vocabulary is a B- or B+-tree [Cutting and Pedersen, 1990].

Some query evaluation regimes also require that the index hold the positions within each document that a term occurs in. Positional information is necessary, for instance, for phrase query processing; but there are also relevance metrics that make use of term proximity even for simple ranked queries. Such information is referred to as *word positions* or *offsets*. An index holding word positions is referred to as a *word-level* index; one holding only document occurrences is referred to as a *document-level* index. Word position information is not used during query evaluation in the system employed for the experiments in this paper, and the indexes used do not contain word positions.

Document pointers within a term's inverted list can be held in a number of orderings. The most obvious is in increasing order of document number. Such an arrangement is referred to as *document-ordered* or *document-sorted*. The document ordering of inverted lists simplifies the adding of new documents to the index. Since the new documents are generally assigned the next document number in sequence, their pointers can be added to the end of the existing inverted lists, without needing to modify the order or content of existing pointers. In addition, document-ordered lists simplify the processing of Boolean and phrase queries, which can be performed as ordered merges upon the inverted lists. However, since users generally only wish to see a listing of the highest-relevance documents, the exhaustive evaluation that document ordering of inverted lists requires can be wasteful.

Figure 2.1 gives a sample of a document-ordered, offsetless inverted index for a fifteen-document collection. The index consists of a vocabulary holding terms and term statistics; the inverted lists, with the on-disk location of a term's inverted list given in its vocabulary entry; and the document map. A selection of terms from the index are shown. The entry for "cat" is read as follows:

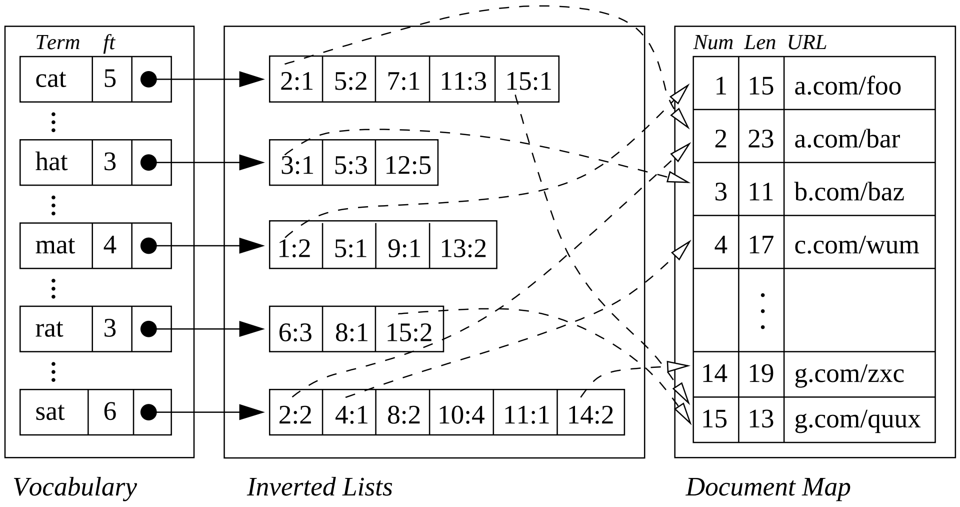


Figure 2.1: Sample document-sorted inverted index.

> The term "cat" occurs in five documents. It occurs once in document number 2, twice in document number 5, once in document number 7, three times in document number 11, and once in document number 15.

An alternative to document-ordered inverted lists is to order lists by decreasing $f_{d,t}$, which is known as *frequency ordering*. Due to the *tf·idf* principle, pointers with higher $f_{d,t}$ values generally have higher $w_{d,t}$ contributions. Therefore, under frequency ordering, processing of an inverted list can be cut short when it is determined that future pointers will not have a significant effect on document rankings [Persin et al., 1996]. An extension of frequency ordering is to pre-compute the similarity contribution or *impact* of each pointer, to store the impact instead of the raw $f_{d,t}$ value in the inverted list, and to order the list by decreasing impact. Such an arrangement is known as *impact ordering* [Hawking, 1998, Anh and Moffat, 2002]. Both frequency ordering and impact ordering allow for greater efficiency in processing ranked queries. However, update is more complicated than for document ordering, as is the processing of Boolean or phrase queries. The system employed for experiments in this thesis uses document-ordered inverted lists.

Inverted list information is typically stored and even processed in a compressed form. The motivation behind the compression of inverted lists is not just to save space on disk. Compressed inverted lists are faster to read off disk, and take up less main memory, allowing for greater caching of data [Witten et al., 1999, Moffat and Zobel, 1996]. Inverted list compression schemes typically use a variable number of bits or bytes to encode numerical values, meaning that smaller values take up less space than larger ones. The $f_{d,t}$ values are typically small, and document numbers in document-ordered inverted lists are encoded as the difference or *gap* from the preceding document number, with the result that they are also, on average, relatively small.

The compression scheme employed by the system used in this thesis is a variable-byte or *vbyte* scheme [Scholer et al., 2002]. In this scheme, the low-order 7 bits in each byte are used to represent a numerical value, whereas the high-order bit indicates whether the byte is the last or *stopper* byte in the encoded number or not. Thus, values up to 128 can be represented in a single byte (0 never occurring, either as a document number difference or as an $f_{d,t}$).

The process of compressing a term's inverted list can best be illustrated with a simple example. Start with the following sample raw inverted list:

$$\langle 15{:}3, 52{:}2, 268{:}1, 670{:}4, 798{:}3 \rangle$$

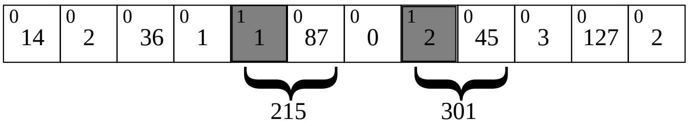
which states that the term occurs three times in document fifteen, twice in document fifty-two, and so forth. Rather than the absolute value of each document number being held, it can be represented as the difference or gap from the previous document number in the list. As the list is document-sorted, the document gap will always be positive. Taking document gaps produces the following list:

$$\langle 15{:}3, 37{:}2, 216{:}1, 302{:}4, 128{:}3 \rangle$$

Because each document occurs in the list only once, and because a pointer is only recorded if its $f_{d,t}$ is at least 1, the value 0 never needs to be encoded. Therefore, each value in the list can be decreased by one before encoding:

$$\langle 14{:}2, 36{:}1, 215{:}0, 301{:}3, 127{:}2 \rangle$$

Finally, these values can be vbyte-encoded:



where each box represents a byte, with the high-order bit shown in the top-left corner and the decimal value of the low-order seven bits show in the center, and with the two multi-byte values bracketed together. Stopper bytes have the high-order bit set to 0 and are unshaded. Whereas the raw inverted list with

32-bit integers would take up 40 bytes, the vbyte-encoded list uses only 12.

A variable-byte encoding scheme does not achieve as good compression as a variable-bit scheme does, but is simpler to implement and provides faster decoding. Using such a scheme, an inverted index can be reduced to around 15% of the size of the original collection [Scholer et al., 2002] if word offsets are included, and less again if word offsets are excluded. The offset-less inverted indexes used in this thesis's experiments are around 4% of the size of the collection. More recently, word-oriented coding schemes have been proposed which combine good compression ratios with excellent decoding speeds [Anh and Moffat, 2005, 2006b]. Other methods are described by Witten et al. [1999].

The evaluation of a query with document-ordered inverted lists can occur in one of two ways, either *document-at-a-time* or *term-at-a-time* [Turtle and Flood, 1995, Kaszkiel and Zobel, 1998]. In document-at-a-time processing, the inverted lists for all query terms are opened (and loaded if memory allows), and processed in a multi-way merge. The similarity score for each document is then fully computed before the next document is considered. Document-at-a-time processing has the advantage that final scores for each document are known immediately; these scores can then be used in a fairly straightforward way to prune out documents that are not going to make the top r results requested by the user. Document-at-a-time processing has the disadvantage that all inverted lists must be read into memory at once, or else multiple reads must be made for each list. There is also some additional computational complexity in performing the multi-way merge.

In term-at-a-time processing, the inverted list of each term is processed in turn. Term pointers create or add to partial relevance scores for the documents they reference; these scores are not complete until the final query term is processed. The partial document relevance scores together with their document identifiers are referred to as *accumulators*. For document-ordered inverted lists, the most suitable data structure to hold the accumulators in is also a document-ordered list. Then, each inverted list is merged with the existing accumulator list, adding to the partial scores of documents for which pointers have already been encountered, and creating new accumulators for documents encountered for the first time. Term-at-a-time processing has the advantage that only one inverted list has to be held in memory at one time; it is the processing model employed in this thesis.

As mentioned, term-at-a-time processing requires the maintenance of a list of accumulators, holding partial document scores. For queries that include frequent terms, the list of accumulators can grow very large. Potentially, as many accumulators can be created as there are documents in the collection. The largest experimental collection employed in this thesis has around 25 million

Choice	Possibilities	In this thesis
index level	document-level, word-level	document-level
list sorting	document, frequency, impact-sorted	document-sorted
compression	vbyte, vbit, word-oriented	vbyte
processing mode	term-at-a-time, document-at-a-time	term-at-a-time
query pruning	quit, continue, dynamic	dynamic

Table 2.1: Implementation options for a full-text query evaluation engine.

documents. Allowing 8 bytes for each accumulator, an accumulator set with as many items as documents in the collection would require 200MB of memory. Such a large memory footprint eats into the memory available for file caching, reduces the capacity for parallelised processing of queries, and increases the processing load of memory management and comparisons during the merging of accumulator and inverted lists. The situation is particularly wasteful given that the user typically requires as few as the top 20 results, and rarely more than the top 1,000.

Various techniques have been proposed in the literature for *pruning* the number of accumulators used in term-at-a-time query evaluation. The best documented are the *Quit* and *Continue* strategies described by Moffat and Zobel [1996]. Both of these strategies work with the idea of an *accumulator limit*. Accumulator growth is unconstrained until the accumulator limit is hit, meaning that each new document occurrence creates a new accumulator; this is referred to as OR-mode processing. In the *Quit* strategy, once the accumulator limit is hit, processing stops, and the existing accumulators are immediately returned for the top r results to be extracted. In the *Continue* strategy, processing continues, but in what is referred to as AND-mode. In AND-mode, pointers for documents with existing accumulators add to the scores of those accumulators, but pointers for newly seen documents do not create new accumulators. Moffat and Zobel [1996] report, from experiments on a collection roughly 2 GB in size, that an accumulator limit of around 1%-5% of the total number of indexed documents leads to retrieval effectiveness as good as that achieved if accumulator growth is unconstrained. The issue of accumulator pruning is revisited in Chapter 5, where a new pruning mechanism based around a dynamic threshold is proposed, and shown to achieve better performance than the existing strategies.

Table 2.1 summarises the different choices to be made in implementing a full-text query evaluation engine, the options for each choice, and the option chosen for this thesis. The query evaluation system employed in this research holds a document-level index with document-sorted inverted lists compressed using a vbyte encoding scheme. Query processing is performed according to the term-at-a-time method, with the accumulators being pruned using a new dynamic query pruning strategy.

The software used in this thesis is a customised version of the Zettair information retrieval system, developed at RMIT University in Australia (see `http://www.seg.rmit.edu.au/zettair`), which incorporates all of the implementation choices described above. Modifications were made to support parallelised processing and the distributed modes of operation discussed in Chapter 3.

2.4 Collections and standards

The goal of the document and query models and relevance metrics introduced in Section 2.2 is to return the documents most relevant to a user's query. How well a given method achieves this goal is referred to as its *retrieval effectiveness*. Assessing retrieval effectiveness is ultimately subjective; the user posing the query must decide whether the search engine has answered the query, and since the query is only an approximate representation of the user's underlying information need, this assessment will differ between different users and even at different times for the one user. Nevertheless, for research purposes, some readily calculable, comparable, and repeatable measure of retrieval effectiveness is essential, in order to assess and compare the relative merits of different query evaluation methods.

Retrieval effectiveness metrics generally require three items of data: a standard document collection; a standard set of queries or *topics* to be run against that document collection; and a standard set of judgments, created by human assessors, as to which documents in the collection are relevant to each query. The latter are sometimes known as *qrels*. For large collections, it is generally not feasible to assess every document for relevance to every query. Instead, *pooling* is commonly used, where for each query the documents retrieved by different systems to some depth r are pooled together, and only these documents are assessed for relevance. Pooling is likely to leave many relevant documents undiscovered; however, it does allow for a fair comparison of effectiveness between different retrieval systems, even if a retrieval system has not contributed documents to the pool, provided pooling is done to an adequate depth [Zobel, 1998].

Relevance judgments may be binary, assessing documents as either wholly relevant or wholly irrelevant to a query, or they may assign degrees of relevance, for instance by distinguishing between highly and partially relevant documents [Voorhees, 2001]. The effectiveness metric then compares the answers produced by the search engine against the relevance judgments or qrels for the collection and query set, and generates an effectiveness score for the run [Harman, 1993]. Where degree of relevance has been assigned in the qrels, this may either be collapsed to binary relevance at some relevance threshold, or an effectiveness metric that incorporates degree of relevance may be used [Kekäläinen and Järvelin, 2002]. In this thesis, binary relevance judgments are employed.

There are two basic metrics for measuring retrieval effectiveness at some cutoff level r of retrieved documents. The first is *precision*: what proportion of the documents returned by the system are relevant? The second is *recall*: of all the relevant documents existing in the collection, what proportion does the system actually return? Let $\mathcal{R}$ be the set of relevant documents in the collection, and $\mathcal{A}_r$ be the set documents retrieved by the system to rank r; then:

$$\begin{aligned}\text{Precision} &= |\mathcal{R} \cap \mathcal{A}_r| / |\mathcal{A}_r| \\ \text{Recall} &= |\mathcal{R} \cap \mathcal{A}_r| / |\mathcal{R}_r|\end{aligned}$$

There are numerous variants on these basic measurements, measuring to different numbers of returned documents and with different biases towards higher-ranked results. One of the most commonly used effectiveness metrics, and the one employed in this thesis, is *mean average precision*, or MAP. Mean average precision is calculated as follows. For each query, consider all relevant documents returned in the ranked top r results, where $r = 1{,}000$ is a typical value. For each relevant document returned, calculate the precision at that document's rank. For each relevant, but not retrieved, document, assign a precision of 0. Then find the average of all these precisions for the query. Finally, calculate the mean of all these precision averages across the query set. The MAP metric incorporates both precision and recall elements, and rewards returning more relevant results higher up the ranking.

To illustrate the calculation of average precision for a given query q, let $\mathcal{D}$ be the collection of documents, $\mathcal{R}$ the set of documents in $\mathcal{D}$ that are relevant to query q, and $\mathcal{A}$ the ranked set of documents retrieved by the system in answer to query q. Assume, without loss of generality, that

$$\begin{aligned}\mathcal{R} &= d_3, d_7, d_{12}, d_{18}, d_{21}, d_{38} \\ \mathcal{A} &= d_{10}, \mathbf{d_7}, \mathbf{d_{21}}, d_1, \mathbf{d_3}\end{aligned}$$

where $\mathcal{R} \cap \mathcal{A}$ has been highlighted within $\mathcal{A}$. The answer set $\mathcal{A}$ holds relevant documents at ranks 2, 3, and 5. The precisions at these ranks are 0.50, 0.66, and 0.60; the unretrieved documents d_{12}, d_{18}, and d_{38} are assigned precision 0. The average precision of $\mathcal{A}$ is therefore $(0.50+0.66+0.60+0.00+0.00+0.00)/6 = 0.29$.

An obvious problem with effectiveness metrics that incorporate recall, including MAP, is that not all relevant documents may have been identified. This is the case for large collections where document pooling has been used. To discover even a large proportion of relevant documents, pooling must be done to an adequate depth, and requires a reasonable number of contributing systems [Zobel, 1998]. However, Sanderson and Zobel [2005] argue that shallower pooling with a larger number of queries is a statistically more effective use of assessors' time. Additionally, the user model underlying MAP is not convincing; in most circumstances, a user's satisfaction at discovering relevant documents is in no way affected by the existence of unreturned relevant documents of which the user is not even aware. Finally, MAP typically evaluates rankings to a depth of 1,000, whereas in the majority of web searches the user does not look beyond the fourth document in the ranking [Joachims et al., 2005]. In this light, a new effectiveness metric called *rank-biased precision* or RBP is proposed by Moffat and Zobel [2006]. RBP does not incorporate a recall component, and therefore for a given evaluation depth r is unaffected by the existence of (known or unknown) relevant documents beyond depth r. Nevertheless, MAP is the metric used in evaluating effectiveness throughout the current study. Our focus here is on efficiency, and effectiveness results are given to quantify the extent (if any) to which efficiency gains are degrading retrieval effectiveness. The fact that MAP performs deep evaluations, and is in widespread use in the IR community, make it a suitably conservative choice for our purposes.

Effectiveness metrics are of limited use unless standard collections, query sets, and relevance judgments are created and made available to the research community. The most active and influential effort to provide such standard experimental data is the annual TREC conference, sponsored by the National Institute of Standard and Technology in the US [Voorhees and Harman, 2005] (see http://trec.nist.gov). TREC has been running since 1992, and has created a number of different text corpora and accompanying query sets and qrels over that time. The largest of these collections to date is the 426 GB GOV2 collection, used in the Terabyte track of TREC. The GOV2 collection was crawled in late 2004 exclusively from websites in the .gov domain. The collection consists of roughly 25 million documents, most of which are HTML web pages, with the remainder being PDF documents converted to plain-text format. The GOV2 collection and subcollections drawn from it are used in the experiments in this thesis. The TREC effort has also produced two standard query sets to accompany this collection, TREC topics 701-750 and 751-800, and corresponding qrels for these query sets. The former query set, and the corresponding qrels, are used for effectiveness measurements in this thesis.

Chapter 3

Distributed information retrieval

Chapter 2 introduced the field of information retrieval, and described the indexing and processing regimes of an information retrieval engine. In this chapter, we examine techniques for distributing the processing of an IR system across multiple machines, something that is essential to support web-scale systems. The first section examines the workload placed upon web-scale search engines, and introduces the concepts of partitioning and replication. In the second and third sections, the two standard partitioning strategies described in the literature, namely document partitioning and term partitioning, are presented. The fourth section introduces a new architecture, pipelining, which aims to combine the best characteristics of document and term partitioning. Previous research in the area of distributed information retrieval is reviewed in the fifth section. Finally, the sixth section provides an overview of the remainder of the thesis, summarising the original research contributions to be presented in Chapters 4, 5, 6, 7 and 8.

3.1 Supporting web-scale workloads

Using the optimisation techniques described in Chapter 2, a single-machine or *monolithic* information retrieval system is able to support impressive workloads, even running on commodity PC-class hardware. Experimental results given in Chapter 6 show that the Zettair search engine, running on such hardware, can process over 15 queries per second against a 426 GB collection. Impressive as the throughput achievable by a monolithic IR system is, however, it is a small fraction of the workload that a contemporary web-scale IR system is required to support. A recent study estimated that as of 2005, there were over 11.5 billion indexable documents on the web [Gulli and Signorini, 2005]. At an estimated 20 KB per document, this equates to something in the order of a quarter of a petabyte of text. Even three years ago, Google was described as handling thousands of queries per second [Barroso et al., 2003], which for such a large amount of indexed text equates to a workload some hundreds of thousands times greater than is supportable by a single PC-class machine. And the workload is growing faster than the capacity of individual machines to support it [Levene and Poulovassilis, 2001].

Due to the workload that must be supported, the implementor of a web-scale search engine is forced to distribute query processing across multiple machines. There are two basic ways in which this can be done. First, the index can be partitioned into subindexes, each managed by a separate machine, leaving each machine with a smaller index to handle. And secondly, the index can be replicated across multiple machines, and the query stream split between these machines, with the result that each machine has to handle fewer queries per time interval. Partitioning and replication can also be combined, for instance by first partitioning the index amongst a cluster of a given number of machines, and then replicating that cluster—which is, in fact, is the standard mechanism for handling workload in web search engines [Barroso et al., 2003]. A number of studies have examined the performance trade-offs between partitioning and replication [Cahoon and McKinley, 1996, Orlando et al., 2001, Cacheda et al., 2004]. Whole-of-index replication is not examined in the current research. Instead, this thesis is concerned with different methods of partitioning and architectures for performing distributed query evaluation upon partitioned indexes.

3.2 Document partitioning

One way of partitioning an index into subindexes is to assign a subset of the documents in the collection to each subindex. Partitioning can be performed by building a monolithic index and splitting up its inverted lists into the designated subsets by document ids. Alternatively, subcollections of the full document collection can be created, and a separate subindex built for each subcollection, by the node that is to host that part of the index. Whichever way partitioning is performed, the resulting subindexes are largely autonomous, capable by themselves of answering queries upon the subcollections they manage. Partitioning an index into document subsets is referred to here as *document partitioning*. It is also referred to in the literature as the *host index* [Tomasic and García-Molina, 1993] or *local index* [Ribeiro-Neto and Barbosa, 1998] organisation, in that each

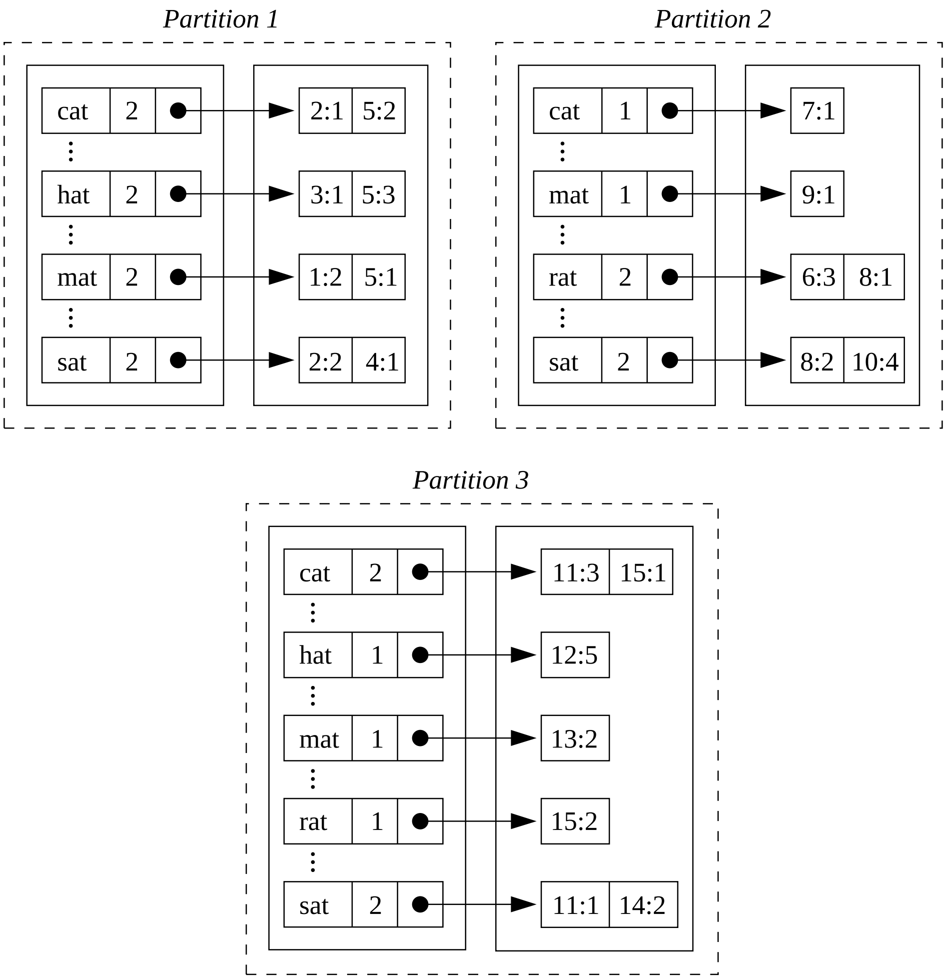


Figure 3.1: A document-partitioned index.

host (node) in the cluster locally holds its own, largely standalone index.

Figure 3.1 illustrates a three-way document partitioning of the sample inverted index introduced in Figure 2.1. In this partitioning, Partition 1 holds information for documents 1 through 5, Partition 2 for documents 6 through 10, and Partition 3 for document 11 through 15. Each partition also holds a local vocabulary, containing information for all terms that occur in the document subcollection it indexes. If a term occurs in more than one partition, as will be the case with reasonably-sized collections for all except the rarest terms, then it will have more than one vocabulary entry. Similarity metrics that employ per-document statistics (such as normalisation by document length) will also have document maps hold these statistics in each partition (not shown in Figure 3.1 for space reasons).

Given a cluster with a document-partitioned index, query processing can be performed as follows. Each node holding a partition is referred to as an *evaluator* node. One node takes on the responsibility of receiving and responding to user queries; it is called the *receptionist* node. The receptionist could be a distinct node, such as the controller node of the cluster, as is the case in the system implemented for this thesis. Alternatively, one of the evaluator nodes could also perform the role of receptionist, or it could be a task running on each of the evaluators, with incoming queries distributed in round-robin fashion. Whichever of these models is chosen, the user's query arrives at the receptionist, which broadcasts it to the evaluator nodes. Each evaluator node evaluates the query against its local index, and returns the local top r results to the receptionist. The receptionist then merges these results, and returns the global top r results to the user. This method of query processing upon a document-partitioned index will be referred to as *document distribution*.

In the document-distributed architecture, each local index could evaluate the query in an entirely autonomous way, using only locally-held values for statistics (such as f_t) to compute the particular similarity metric being used. However, the use of local statistics can lead to a degradation in retrieval effectiveness, since relative term frequencies in a subcollection may not be the same as in the full collection [Ribeiro-Neto and Barbosa, 1998]. Alternatively, global term statistics could be distributed by the receptionist. For instance, along with each query, the global f_t values for the terms in the query could be sent. The receptionist could collect and aggregate local statistics from evaluator nodes at query evaluation time, or else could hold global term statistics in its own vocabulary. The latter is the implementation method used in this thesis.

Document partitioning has the great benefit of simplicity, in design, implementation, and maintenance. Since each subindex is largely autonomous, it is, for instance, trivial to turn a monolithic search engine system into a document-distributed one, especially if local measures are used instead of global ones. Query processing is performed in a highly parallelised way, giving excellent average response time at low loads.

3.3 Term partitioning

An alternative to partitioning the index by documents is to partition it by terms. Under such a partitioning, each partition holds a subset of the global vocabulary. The partition holds all information for each term in its vocabulary subset, the full inverted list in particular. However, it holds no information on terms outside of its vocabulary subset. This method of partitioning an index is referred to here as *term-partitioning*. Elsewhere in the literature it is referred to as the

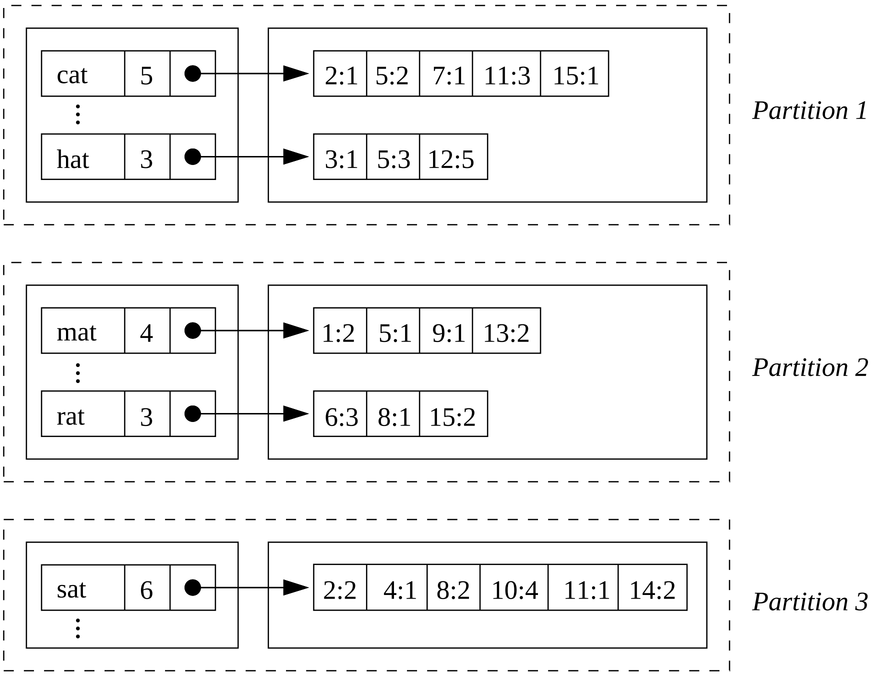


Figure 3.2: A term-partitioned index.

system index [Tomasic and García-Molina, 1993] or *global index* [Ribeiro-Neto and Barbosa, 1998] organisation.

Figure 3.2 shows the same inverted index as Figure 3.1, this time term-partitioned across three nodes. Partition 1 holds the information for terms "cat" and "rat", Partition 2 for terms "mat" and "rat", and Partition 3 for the term "sat". The inverted list for each term is held contiguously, and each term requires only a single vocabulary entry.

A number of strategies are possible for the construction of a term-partitioned index, all of them more complex than for a document-partitioned one. First, a monolithic index could be built, and then statically partitioned. Second, multiple passes could be made over the collection, each extracting only terms occurring in a given subcollection. Third, a central machine could parse the collection and dynamically distribute index information to nodes responsible for each partition. Or fourth, each node could build an index for a subcollection, as when building a document-partitioned index, and then they could exchange term information between them [Ribeiro-Neto et al., 1999].

Query processing on a term-partitioned index can also be performed in a variety of ways. The model examined in this thesis involves a single *processor* node, which performs the query evaluation, and a number of *librarian* nodes, which store the subindexes. When a query arrives at the processor, it is parsed into its constituent terms. For each query term, the processor sends a fetch request to the librarian whose subindex holds that term's information. The librarian loads the information for the query term, including its inverted list, and returns it to the processor. The processor then evaluates the query, using the term information shipped from the librarian nodes, and returns the top r results to the user [MacFarlane et al., 2000]. This processing model is referred to here as *term distribution*.

The main advantage of a term-partitioned index lies in the better contiguity of data on disk compared to document partitioning [Badue et al., 2001, Moffat and Zobel, 2004]. In a document-partitioned index with k parts, most inverted lists are broken up into k fragments. Thus, evaluating a query of $|q|$ terms requires $k \cdot |q|$ distinct disk reads to load inverted list information, each carrying with it the overhead of a disk seek—assuming, that is, that the inverted lists are not already cached in memory. On a busy system with an index large enough to be mostly disk-resident during processing, the large number of disk seeks has the potential to be a major performance bottleneck. In contrast, if the index is term-partitioned, inverted lists remain unfragmented, and the evaluation will require only $|q|$ distinct disk reads.

Term partitioning also offers advantages through its smaller per-node vocabulary. The vocabulary size advantage over document partitioning is muted when only the on-disk or latent vocabulary is considered, since the vocabularies of different document subsets differ substantially, due to the ongoing occurrence of new (predominantly non-dictionary) terms [Williams and Zobel, 2005]. However, if the vocabulary actually accessed during query processing is considered, then term partitioning's advantage is more pronounced. Each node in a document-partitioned system accesses a run-time vocabulary almost as large as that of a monolithic index, whereas in a term-partitioned system each node accesses and caches only a fraction of this vocabulary.

There are three problems with the term-distributed architecture sketched above. The first is that all query processing is performed by the receptionist. The other nodes in the cluster only perform the processing-light (albeit disk-heavy) tasks of vocabulary lookup and inverted list loading, and are, in a sense, little more than inverted list servers. In such an architecture, the receptionist can easily become a bottleneck, with the other nodes being underused. The cluster nodes could be given more work to do, for instance by replacing a simple document-ordered inverted list processing regime with one involving dynamic frequency- or impact-based filtering [Badue et al., 2001, Tomasic and García-Molina, 1993]. Additionally, processing could be cycled between the cluster nodes, rather than fixed on a single receptionist. However, the task of balancing

CPU workloads between cluster nodes and the receptionists would continue to be problematic. The second issue of concern is the bandwidth required to ship inverted lists to the receptionists. This, too, might become a bottleneck, and even if it did not, the processing involved in reading so much data off the network adds to the burden upon the receptionist. And the third problem is that partitioning the collection by terms can lead to load imbalance, as the distribution of term occurrences in both collections and query streams is highly skewed [Tomasic and García-Molina, 1993].

3.4 Pipelining

The problem of a processing bottleneck at the receptionist in the term-distributed architecture provides the inspiration for an alternative model for query evaluation using a term-partitioned index. Instead of shipping all of the data to a central processor, the new model ships the processing state to the data. Upon receipt of a user query, the receptionist constructs a *query bundle* to encapsulate the query and its processing state. The query bundle consists of the query itself; a routing table of the nodes holding terms in the query, sorted in processing order; and an (initially empty) accumulator structure. The query bundle is then shipped amongst the evaluator nodes. On arrival at a node, the accumulators are extracted from the bundle, and reconstituted in memory. Terms held on that node are evaluated as normal. If the query has more nodes to visit, the updated accumulators are serialised into the query bundle, and the bundle is sent on to the next evaluator node. If, however, the current node was the last node in the routing table, the top r results are extracted, and sent to the receptionist, who returns them to the user. The routing of the processing state through a sequence of processor nodes inspires the name of the architecture, which is *pipelined distribution* or simply *pipelining*.

Figure 3.3 illustrates the routing of a query through a pipelined system. The index has been term-partitioned amongst 8 nodes. The query contains four terms, which in order of increasing f_t are denoted as t_1, t_2, t_3, and t_4. Term t_1 is held on node 3, t_2 and t_3 on node 6, and t_4 on node 7. The receptionist creates a query bundle containing the query and the routing information "node 3 : t_1; node 6 : t_2, t_3; node 7 : t_4". The query bundle is sent first to node 3, which loads the inverted list for t_1, and processes it to create an initial set of accumulators. Node 3 adds these accumulators to the bundle, and ships it to node 6. Node 6 extracts the accumulators, loads the inverted lists for terms t_2 and t_3, and processes the lists against the existing accumulators to create an updated set of accumulators. The updated accumulators are added to the query bundle, which is then shipped to node 7. On receipt, node 7 extracts the

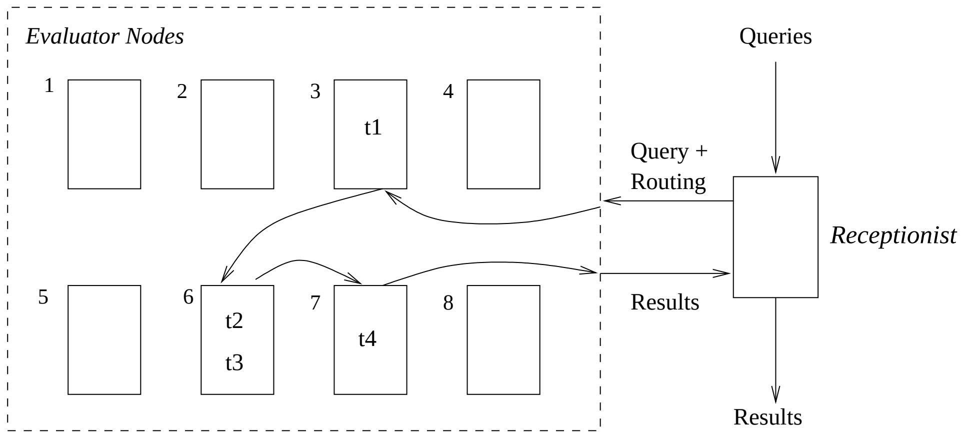


Figure 3.3: Routing of a query through a pipelined system.

accumulators, loads the inverted list for term t_4, and processes the list against the accumulators, completing the inverted list processing stage. Finally, node 7 extracts the identifiers of the documents with the highest r similarity scores, and ships them to the receptionist, to be returned to the user.

The routing order of the query bundle needs to follow the processing order specified by the query pruning regime being employed, which is typically by increasing f_t. For this reason, the receptionist must keep a vocabulary holding at least this information, whereas in the term-distributed architecture the receptionist does not require such a vocabulary. Other than this, the pipelined architecture runs on top of the same term-partitioned index that term distribution does.

Pipelining shares the same disk access and vocabulary size benefits that term distribution has over document distribution. In addition, by sharing processing amongst the nodes, pipelining removes the processing bottleneck at the receptionist node. As will be demonstrated in Chapter 5, the use of accumulator pruning and compression means that pipelining also uses less bandwidth than the term-distributed architecture.

3.5 Previous research

Previous work in the area of distributed information retrieval can be roughly divided into three categories: simulated studies of document and term distribution; empirical studies of the two architectures; and alternative architectures and approaches.

3.5.1 Simulated studies

A number of studies of distributed information retrieval architectures have previously been undertaken. One of the earliest comparative studies is that of Tomasic and García-Molina [1993], who compare document and term partitioning. They build an analytical model of the evaluation of conjunctive Boolean queries. Simulations of the different architectures are then constructed from the model, and these simulations are used to determine each architecture's performance characteristics. In the term-distributed system, Tomasic and García-Molina investigate "pre-fetching", which involves retrieving the results for the lowest-frequency term or single-machine sub-query. These partial results are then sent to the remaining nodes, which intersect them with their local lists before returning the combined results to the receptionist. The idea is to reduce network bandwidth, and it has some similarities with pipelining, although as described it can only be applied to Boolean queries. Tomasic and García-Molina find that term-wise distribution greatly underperforms document distribution due to network saturation.

The method of simulation based upon an analytical model has been followed by a number of other researchers, generally with analytical models similar to the one employed by Tomasic and García-Molina, which accounts for costs in disk access, network, and (in a simplified way) processing. However, their analytical model does not include caching effects, but assumes that all inverted list information must be loaded from disk, and (implicitly) that all vocabulary information is held in memory and incurs no access costs. A later study by the same authors does introduce a simple caching model, but it is not integrated into the main body of simulated results [Tomasic and García-Molina, 1996]. The later study finds that with shorter inverted lists and a faster network, term-wise distribution can outperform document distribution.

The system performance characteristics used to derive the analytical model built by Tomasic and García-Molina [1993] are largely arrived at by rough estimation, and where they are based upon actual benchmarking, are from outside the IR domain. In contrast, Cahoon and McKinley [1996] take the monolithic InQuery system, and measure the resources used for each of the operations involved in query evaluation. These measurements are used to build a simulation model. Again, however, caching and memory effects are not modelled, and it is assumed that all inverted list data is disk resident. The document-distributed architecture is investigated for clusters ranging in size from 1 to 128 machines, with a focus on the balance between clients, query brokers, and servers. Term-wise distribution is not considered. This work is extended in Cahoon et al. [2000] to include multi-threading. In turn, Cacheda et al. [2004]

adopt a similar methodology to explore the trade-offs between partitioning, replication, and topic-specific clustering, but again only examining document partitioning. The trade-offs between scaling and replication are explored in an implemented system by Orlando et al. [2001], although some of their findings are counter-intuitive, such as that a two-node replicated system is more than twice as fast as a single-node unreplicated one, which suggests that something is amiss either with their implementation or with their experimental methodology.

Measurements from a real system also form the basis for the analytical model behind the simulation used in Ribeiro-Neto and Barbosa [1998] to compare the document and term-distributed architectures. Here, in contrast to Tomasic and García-Molina [1993], ranked query evaluation is modelled. Frequency-ordered, rather than document-ordered, inverted lists are used [Persin et al., 1996]. As a result, in the term-distributed architecture, the full inverted list is not returned to the receptionist, but only the highest-frequency pointers, reducing both bandwidth usage and processing load on the receptionist. The simulation shows term distribution outperforming document distribution, especially as the number of machines in the cluster grows.

The work of Jeong and Omiecinski [1995] is concerned with a single, multiprocessor and multi-disk system, rather than with distribution across multiple machines. Nevertheless, it performs a simulated comparison between document and term partitioning, in the context of multiple disks. Jeong and Omiecinski explore in detail the issue of the I/O load imbalance that term partitioning causes, due to inverted list lengths varying between terms. They explore different techniques for evening out I/O load, including balancing based on both inverted list size and access frequency. These approaches are similar to the term-allocation workload balancing techniques examined in Chapter 7, except that only I/O load is considered, and the environment is a multi-disk rather than a multi-machine one. Jeong and Omiecinski find that load balancing is a significant issue in a term-partitioned system, and not one that is easily solved. Their work examines skew in the distribution of query term occurrences, which is linked to load imbalance in a term-partitioned system. They present uniform and skewed query models, and examine the effect of different degrees of skew. However, in their skewed model, the frequency ranking of terms in the query set follows the frequency ranking of terms in the collection (that is, the most frequent term in the query set is also the most frequent term in the collection), whereas in real data sets, the correlation between the two is weak [Tomasic and García-Molina, 1993, Baeza-Yates, 2005]. Again, no account is taken of caching effects.

A summary of earlier work on parallel processing and information retrieval can be found in Rasmusen [1991].

3.5.2 Empirical studies

The comparisons between document-distribution and term-distribution in the earlier papers rely on simulation, whereas recent studies have involved experiments on implemented systems. One such investigation is that of MacFarlane et al. [2000]. They find that the document-distributed architecture scales well with cluster size, whereas term distribution fails to scale at all, and underperforms even on small clusters. However, there are a number of deficiencies in their experimental setup. Only 50 queries are used, which is too small a sample to generalise results from, and also too short a run to properly explore caching and parallelisation issues. A fixed collection of 1 GB is split amongst between 2 to 7 machines, meaning that as the cluster scales, the per-machine index shrinks; it would be preferable instead (or in addition) to maintain per-machine collection size as the cluster scales. Also, since each machine in the cluster has 128 MB of main memory, the index can fit in main memory, even on a two-machine cluster; however, it appears that experimental runs are made "cold," thus leaving a confusing mix of caching and disk effects. For instance, MacFarlane et al. find that a document-distributed system scales better in cluster size with whole-topic queries than with title-only ones. This is possibly an artifact of the fact that whole-topic queries contain more repeated terms across the query set, which means better caching when starting from a cold run; this in turn helps offset the greater on-disk fragmentation that occurs with a higher degree of document partitioning.

While MacFarlane et al. [2000] found in favour of the document-distributed architecture, the implemented experiments of Badue et al. [2001] show term distribution outperforming document distribution. The system implemented is similar to that simulated in Ribeiro-Neto and Barbosa [1998]. In particular, inverted lists are frequency-ordered, and only part of each list is returned from the holding node to the receptionist. While this is certainly a desirable setup from the point of view of the efficiency of term distribution, the complexity of updates to a frequency-ordered index mean that document-ordering remains a pragmatic choice, and the performance of term distribution with document-ordered lists remains in question. In addition, the study of Badue et al. has some methodological problems, in particular with the two query sets employed. The first query set is 50 TREC queries, which is too small a query set to safely generalise from. The second set consists of 2,000 artificial queries generated by random choice from the collection vocabulary. Random choice results in a uniform distribution of query term occurrences, which the Badue et al. themselves acknowledge artificially improves load balancing.

3.5.3 Other approaches

A quite different architecture referred to as "hybrid partition inverted files" is presented by Xi et al. [2002]. Their approach incorporates aspects of both term-partitioning and document-partitioning. Processing of lists is performed by the receptionist, as in the term distributed architecture described above. However, inverted lists are divided into fixed-length chunks, which are distributed amongst the cluster nodes. The goal is to balance the workload involved in loading data off disks and transferring it back to the receptionist. The hybrid partition inverted files architecture does not tackle the two main issues of the term-distributed system, namely the processing bottleneck on the receptionist and the amount of bandwidth required to ship inverted lists to the receptionist. Moreover, the fragmenting of inverted lists removes the contiguity of data that is term partitioning's main advantage, resulting in a "worst-of-both-worlds" setup.

The terms "distributed information retrieval" are also used to refer to a different problem domain, namely the aggregation of logically distinct and often organisationally and locationally separate text databases, connected over a wide-area network, such as the Internet. *Federated search*, as the problem domain is also termed, is concerned with choosing which index or indexes a query should be sent to, and merging the results that the different indexes produce [Callan, 2000, Callan et al., 2003, Si and Callan, 2005]. Some interesting research is presented by Lu and McKinley [1999] upon using concepts developed in federated search, such as index specialisation and formulae for weighting and merging results from disparate indexes, to build partial replicas specialised for common user query groups or "topics" as part of a LAN-distributed index. These specialised replicas are intended as a more flexible form of answer caching (see also Lu and McKinley [2000] and Cacheda et al. [2004]). However, federated search is beyond the scope of this thesis.

3.6 Overview of research contribution

The above sections have detailed the current state of research into distributed information retrieval architectures, and provide the background to the investigation described in the following chapters.

First, in Chapter 4, a methodology for performing rigorous and scalable experiments upon implemented distributed query evaluation systems is developed and described. The aim is to avoid the methodological deficiencies of previous experimental regimes, as well as to provide a more realistic experimental framework than those offered by simulation based upon analytical models. Attention needs to be paid to a number of issues in developing such a methodology. These include the scalability of the experimental results, the use and degree of processing parallelism, the makeup of the query set, and controlling disk access performance variability.

As has been described, the pipelined architecture ships the processing state, consisting chiefly of accumulators, between nodes in the cluster. It is therefore essential for the efficiency of the pipelined system that accumulator growth is contained, and that accumulators are efficiently compressed for shipping. These issues of accumulator handling are addressed in Chapter 5.

Next, the methodology developed in Chapter 4 is applied, and an extensive experimental assessment of the document-distributed, term-distributed, and pipelined architectures is carried out, with results presented in Chapter 6. Suitable settings for variables such as accumulator limits and the degree of parallelism are established experimentally. Then, with these settings decided, timed runs are made on each of the three distributed architectures for clusters of 2, 4, and 8 machines, and additionally on a monolithic baseline. These setups are run against collections scaled homogeneously from a full 426 GB down to 1/64th of this size, allowing for the exploration of several different varieties of scalability. The results demonstrate that a term-distributed architecture with document-ordered inverted lists and a centralised processing node faces a severe bottleneck in processing, and as a result performs poorly, failing to scale at all with cluster size. The results also show that pipelining performs and scales much better than term distribution, but still falls well short of document distribution. Analysis of these results demonstrates that, while pipelining achieves the disk I/O benefits it promises, it is hobbled by serious workload imbalance.

The following chapter, Chapter 7, explores the issue of load balancing in the pipelined architecture in detail. Load imbalance is shown to derive from the skewed distribution of inverted list lengths and access frequencies between different terms. Methods for assigning terms to nodes in order to even out load are considered and evaluated, both in simulation and in actual runs. Such techniques are demonstrated to narrow the performance gap with document-distribution, but some load imbalance remains, and so the full potential of pipelining's I/O and memory usage benefits is still not achieved. A technique for replicating high-workload terms on more than one node is therefore proposed. Different degrees and depths of replication are examined, and the differing routing strategies that replication enables the system to employ are explored. It is demonstrated that, with partial term replication and intelligent routing of queries, the pipelined system's load imbalance problems are substantially resolved, and the architecture is able to achieve marginally higher throughput than document distribution.

Chapter 8 then takes the improved pipelining strategy devised in Chapter 7 and demonstrates its scalability. In addition, the partially-replicated pipelined system is found to substantially outperform document distribution in a more memory-limited environment. On the other hand, an examination of average response times for the two architectures under varying workloads shows that document distribution offers faster and more scalable response times at low to medium workloads.

Finally, Chapter 9 gives an overview of the results of this research, provides advice to the system implementor on which distribution architecture to choose, and outlines directions for future research in this area.

Chapter 4

Experimental technique

Chapter 3 described the differing conclusions of previous research into the relative merits of document and term distribution. These differences underline how important it is to design experiments carefully, especially when investigating the merits of different distributed architectures. We must choose appropriate experimental data; specify what the experimental variables are and which variable is being measured in each experiment; minimise the variability that external factors may cause in the results; and ensure the retrieval correctness of the methods employed even when focused on efficiency. These issues of experimental technique are the subject of the current chapter.

4.1 Queries

Finding a suitable query set is a perennial problem in information retrieval efficiency research. The query set must be large enough to allow a sufficiently long execution period, and an adequate mix of different queries, for efficiency results to be reliable. The set also needs to be either real or realistic, so that what is being modelled in experiments can be applied to what would be experienced in real use. And it should be appropriate to the test collection being used. An "appropriate" query set is, above all, one that is from the same information domain as the collection; legal queries, for instance, are not appropriate for a collection of medical documents. It is also highly desirable that the queries were formulated at a similar time to when the collection was created; queries concerning an event that occurred in 2005 are a poor match for a collection gathered in 1998.

4.1.1 Appropriate and inappropriate query sets

A large and realistic query set is most easily obtained from a real-world search engine's query log. A number of search engine query logs are readily available and have been studied in the literature, such as the Excite97 and AltaVista logs [Jansen et al., 1998, Spink et al., 2001, Silverstein et al., 1999]. However, the logs of whole-of-web search engines may not be appropriate to a particular test collection, particularly if the collection is domain-specific and drawn from a different period than the logs.

Much current research uses the TREC GOV2 collection, described in Section 2.4, which is to date the largest of the TREC collections. Unfortunately, the GOV2 collection is poorly matched to the publicly available search engine logs. First, GOV2 is a domain-specific collection, crawled solely from United States government web sites. And second, the GOV2 collection was crawled in early 2004, whereas most of the readily available real-world query logs date back to the late 1990s. The Excite97 query log, for example, consists of one million-odd queries submitted on a single day in September 1997. The domain mismatch with GOV2 can be seen from the fact that the five most popular queries in Excite97 (which all happen to be single-word queries) are "sex", "yahoo", "chat", "playboy", and "porn". The time mismatch can be seen from the fact that the most popular multi-word query is "princess diana" (the query log dates from ten days after her funeral).

From the viewpoint of efficiency experiments, the semantic appropriateness of the queries to the indexed collection is not directly important. However, inappropriate query sets can have quite different statistical properties from appropriate ones, and this does have a critical impact upon efficiency results. The query terms of an inappropriate log will on average be less frequent in the collection than those of an appropriate query set. As a result, inappropriate queries will on average find fewer matching documents than appropriate queries do. The performance characteristics of a system will, therefore, be different with inappropriate and than with appropriate query sets: not only will the system be able to process more inappropriate than appropriate queries per second, but also the load upon different parts of the system will be different.

The problems associated with an inappropriate query set can be illustrated by taking the Excite97 query log and comparing its statistical properties when applied to the GOV2 collection against those when applied to a collection to which it is (more) appropriate. A suitable collection for the latter purposes is the wt10g collection from TREC, used in the TREC-9 and 2001 TREC Web Tracks. The wt10g collection contains 10 GB of text extracted, with an eye to enhancing coherence and document quality, from the 100 GB VLC2 collection (also known as wt100g)

Term	Collection	
	`wt10g`	`GOV2`
sex	0.0187	0.0139
free	0.1359	0.0670
nude	0.0020	0.0001
pictures	0.0254	0.0060
pics	0.0023	0.0002

Table 4.1: Term f_t as a proportion of number of documents, over the `TREC` *collections* `wt10g` *and* `GOV2`*, for the five most frequent terms in the* `Excite97` *query log.*

[Soboroff, 2002, Bailey et al., 2003]; we choose `wt10g` in preference to `VLC2` due to its improved document quality. The `VLC2` collection itself was crawled in 1997, from the web as a whole, not restricted to any particular set of domains. On both date and domain, therefore, `wt10g` is a better match for `Excite97` than `GOV2`.

Table 4.1 takes the five most frequently occurring query terms from the `Excite97` log and compares the proportion of documents they occur in for each collection. In all cases, the terms are less common in `GOV2` than in `wt10g`, ranging from two-thirds to a twentieth of the frequency. It is therefore to be expected that the query log would take less time (adjusting for collection size) to process against the former, inappropriate collection than against the latter, appropriate one. To test this hypothesis, a subcollection was extracted from the `GOV2` collection containing the same number and similar composition of documents as `wt10g`. Monolithic indexes of both collections were built, and 10,000 `Excite97` queries were run against them. The run took 16% longer against `wt10g` than against the `GOV2` subcollection.

A characteristic of collection inappropriateness that is particularly relevant to distributed experiments is its effect on term workload skew. The *workload* L_t of a term t is the amount of work that t imposes upon the system during the processing of a query set. A term's workload can be measured as the product of the byte length of the term's inverted list in the index, B_t, and the term's frequency in the query set, $\mathcal{Q}_t$. Thus, $L_t = B_t \cdot \mathcal{Q}_t$. Hence, the proportional workload of t in $\mathcal{Q}$ can be defined as:

$$\bar{L}_t = \frac{L_t}{\sum_{t \in T_{\mathcal{Q}}} L_t}$$

where $T_{\mathcal{Q}}$ is the vocabulary of the query set. Now, since both natural collections and natural query sets are well known to have a skewed term frequency distribution [Baeza-Yates, 2005], the workload of a query set's vocabulary will

wt10g		GOV2	
term	weight	term	weight
free	0.062	free	0.035
new	0.028	new	0.031
home	0.018	home	0.022
one	0.015	search	0.017
sex	0.015	sex	0.013

Table 4.2: Proportional workload of five most costly terms for the second 10,000 *queries of the* `Excite97` *log, when run against the* `wt10g` *and* `GOV2` *collections.*

also be highly skewed.

Table 4.2 provides the workload figures for the five highest-workload terms in a run of 10,000 `Excite97` queries against the two collections. Note the much greater term workload skew for `wt10g` when compared to `GOV2`, in particular for the most frequent term, "`free`", which accounts for 6.2% of the total workload when processing the query set against the former collection, but only 3.4% against the latter. Lessened term workload skew in inappropriate query sets is particularly distorting for experiments on term-partitioned indexes, as it directly affects workload balance between nodes. The issue of workload balance is examined in detail in Chapter 7. For now, it is sufficient to note that, intuitively, a random partitioning of terms with higher workload skew is likely to result in greater workload imbalance than a random partitioning of terms with a lower workload skew. Therefore, using an inappropriate query set is likely to create an unrealistically balanced workload.

4.1.2 Synthetic query set generation

Section 4.1.1 has demonstrated the importance of using an appropriate query set in experiments into the efficiency of distributed query evaluation architectures. Ideally, a real-world query log from the same domain and time period as the sample collection should be used. If such a query log is not available, however, it is possible to synthetically generate one with the appropriate statistical properties.

Synthetic query generation is not a new idea in efficiency research. Indeed, a number of previous studies in distributed query evaluation have employed synthetic query sets [Tomasic and García-Molina, 1993, Jeong and Omiecinski, 1995, Badue et al., 2001]. Amongst the attractions of synthetic queries is the possibility of smoothly varying the statistics of a query set in order to see the effect these variations have on performance: to investigate, for instance, what happens if queries are longer or shorter, or contain rare or frequent terms.

However, if realistic statistical properties are to be achieved, synthetic query set generation needs to be undertaken with care. Consider, for instance, a method that generates queries by randomly selecting terms from the index's vocabulary. Such a method produces only incidental skew in the query term frequency distribution, leading to an artificially balanced workload distribution amongst the nodes of a term-partitioned system.

For research into the efficiency of different distributed text query evaluation architectures, a synthetic query set should match an appropriate real-world query log on the following criteria, in decreasing order of importance:

1. Query set term frequency distribution.
2. Collection term frequency distribution.
3. Query length distribution.
4. Query term co-occurrence distribution.

Note that query coherence is not a requirement. That is to say, it does not matter if the synthetically generated queries "make sense" to a human reader. If the experiments involve phrase queries, query coherence might indirectly be important, as only queries making some sort of (syntactic, if not semantic) sense are likely to occur as phrases. However, for the non-conjunctive ranked queries used in all the experiments in this thesis, queries are merely bags of terms with certain statistical properties.

The method proposed to meet the above requirements is one of synthetic query set generation by translation based on term frequency. The generation process works as follows:

We wish to generate a synthetic query set $\mathcal{Q}'$ that is appropriate to a target collection $\mathcal{D}'$. We take an existing query set $\mathcal{Q}$ (the source query set) and a collection $\mathcal{D}$ (the source collection) for which $\mathcal{Q}$ is appropriate. Let $T_{\mathcal{Q}}$ be the set of terms occurring in $\mathcal{Q}$, $N_{\mathcal{D}}$ be the number of documents in $\mathcal{D}$, and $f_{\mathcal{D},t}$ be the number of documents in $\mathcal{D}$ in which term t appears. Similarly, let $T_{\mathcal{D}'}$ be the set of terms occurring in $\mathcal{D}'$. The key step is, for each $t \in T_{\mathcal{Q}}$, to randomly pick a term $t' \in T_{\mathcal{D}'}$ such that $f_{\mathcal{D},t}/N_{\mathcal{D}} \approx f_{\mathcal{D}',t'}/N_{\mathcal{D}'}$. The chosen term t' then becomes the *translation* of t.

The above process maintains identical query length and query term frequency distributions between $\mathcal{Q}$ and $\mathcal{Q}'$, and similar f_t distributions, satisfying the three most important criteria for synthetic query set generation. To also preserve the same query term co-occurrence rate in a deterministic way is very difficult, given that co-occurrence needs to matched up across multiple tuples of terms. Nevertheless, there are some heuristic approaches that provide for reasonably similar co-occurrence rates. The problem can be stated as follows. Term

spice sex	⇒	contra vhs
cartoon art	⇒	proposition claims
star trek	⇒	especially eliminated

Figure 4.1: Sample query translations. `Excite97` *queries are listed on the left; on the right are their* `SYNQ` *equivalents. For instance, the term "sex", which occurs in 1.87% of* `wt10g` *documents but only 1.39% of* `GOV2` *documents, is translated to "vhs", which occurs in 1.78% of* `GOV2` *documents.*

co-occurrence in a query indicates semantic relatedness between the terms, and thus predicts higher-than-random co-occurrence between terms in the collection. However, the synthetic query generation algorithm described above will by itself only provide random co-occurrences. Therefore, it needs to be enhanced with a heuristic to increase the co-occurrence rate.

Assume that translation is performed, and query term co-occurrence resolved, on a query-by-query basis. The problem can then be more formally stated as follows. Each query $q \in \mathcal{Q}$ consists of distinct (possibly repeated) terms $t_1 \ldots t_n$ ($t_n \in T_{\mathcal{Q}}$). At the start of the processing of q, suppose that terms $t_1 \ldots t_i$ ($0 \leq i \leq n$) already have translations $t'_1, \ldots, t'_i$ (arrived at from translating earlier queries), and others $t_{i+1} \ldots t_n$ do not. Let $\Gamma(\mathcal{D}, q)$ be the number of documents in collection $\mathcal{D}$ that all terms in q co-occur in. If $\Gamma(\mathcal{D}, q) \neq 0$, and $\Gamma(\mathcal{D}', t'_1 \ldots t'_i) \neq 0$, then the goal is to choose $t'_{i+1} \ldots t'_n$ such as to heuristically maximise $\Gamma(\mathcal{D}', q')$.

The heuristic method used to boost translated query term co-occurrences is as follows. Process each term $t_j \in \{t_{i+1} \ldots t_n\}$ in turn. For each t_j, randomly select three candidate translations, ${t'_j}^1$, ${t'_j}^2$, and ${t'_j}^3$. Then choose the translation ${t'_j}^x$ which maximises $\Gamma(\mathcal{D}', t'_1 \ldots t'_i, {t'_j}^x)$. If $\forall x, \Gamma(\mathcal{D}', t'_1 \ldots t'_i, {t'_j}^x) = 0$, then continue randomly selecting candidate translations ${t'_j}^y$ until $\Gamma(\mathcal{D}', t'_1 \ldots t'_i, {t'_j}^y) > 0$ or seven more attempts have been exhausted. Then, add t'_j to $t'_1 \ldots t'_i$, and recursively process t'_{j+1}, until all terms in q are translated.

For the experiments reported in this thesis, the target collection $\mathcal{D}'$ was `GOV2`, the source collection $\mathcal{D}$ was `wt10g`, and the source query set $\mathcal{Q}$ was `Excite97`. The synthetic query set $\mathcal{Q}'$ generated by this process is referred to as `SYNQ`. Figure 4.1 shows some sample translations, illustrating the fact that the translated queries generally do not make semantic sense. However, taken with the target collection, they do closely match the source query set, taken with the source collection, in the statistical aspects that are important for distributed efficiency experiments.

4.2 Query parallelism

Enabling query processing parallelism increases the throughput even of a single-processor monolithic system. While one query is waiting for a disk read to complete, computation can continue on another that has its required data in memory. Additionally, modern CPU architectures increasingly support parallelised processing, even on single-chip systems. For instance, the processors used in the experiments for this thesis support hyper-threading; and dual-core CPUs seem likely soon to be ubiquitous.

Document-distributed processing is akin to multiple simultaneous monolithic evaluations, so it gains a similar boost from parallelisation. In addition, query parallelism in a document-distributed system allows for useful work to continue during the latency involved in the distribution of queries and receipt of results over the network; in a strictly serialised system, this would be idle time for individual nodes. Experimental results reported in Chapter 6 show that a threefold increase in throughput can be achieved on an 8-node document-distributed system by allowing maximal query parallelisation.

Parallelisation is even more important for term-distributed architectures. In a serialised term-distributed system, only the nodes that hold information for the terms in each query are involved during the processing of that query; the rest remain idle. If the cluster has k nodes, and the average number of terms per query is $|q|^{\text{avg}}$, and assuming for simplicity that no two terms in any query occur on the same node, then in a serialised term-distributed system, only $|q|^{\text{avg}}/k$ of the system is used on average for each query; and since typical figures for the experiments reported here are $k = 8$ and $|q|^{\text{avg}} = 2.6$, it can be seen that system underutilisation is severe. Indeed, since different terms' inverted lists have greatly differing lengths, the actual underutilisation is even greater than this formulation suggests, as some nodes will finish their work before others and then sit idle for the remainder of the query's processing.

The waste of resources in serialised processing is more severe again for pipelining. A given query bundle can only be at one single evaluator node in a pipelined system at a time. In a serialised pipelined system, therefore, only $1/k$ of the system will be in use at any one time, even without making an allowance for network latency. Experimental results given in Chapter 6 show that on an 8-node system, maximal parallelisation allows pipelining to achieve more than seven times the throughput of a serialised system.

For this reason, the efficiency experiments in this thesis are performed in an environment that allows parallelised processing. The implications of this for performance calculation and experimental methodology are examined in the following subsections.

4.2.1 Throughput versus average response time

There are two main measures of the performance of an IR system. The first is throughput, the average number of queries the system is able to process in a unit of time. The second is average response time, the average elapsed time between the receipt of a query by the system and the dispatch of the query's result to the user. If query processing is serialised, and assuming that there is no delay between one query completing and the next one starting, then the relationship between throughput and average response time is a simple one: throughput is the inverse of average response time. More formally, let N_Q be the number of queries processed, and s be the amount of time required to process them; then

$$\tau = N_Q/s$$

is the system's throughput, and

$$\rho = s/N_Q = 1/\tau$$

is the average response time.

If parallelised processing is allowed, the relationship between throughput and average response time is not so straightforward. If more than one query is processed at the same time, throughput benefits from the parallelisation: for instance, computation for one query can continue while another query is waiting on disk access. Nevertheless, some contention for resources will occur, and so the elapsed time to process each query will increase. As a result, there is a trade-off between throughput and average response time.

In a real-world system, queries arrive at varying intervals, and the number of queries in process at any given time is also variable, making the formulation of the relationship between throughput and average response time complex. In an experimental environment, in contrast, a situation can be enforced in which the number of queries in process at any given time is kept constant; as soon as one query currently in the system finishes, another starts. The experimentally-fixed number of queries simultaneously in process in the system is referred to here as *simulq*. The relationship between throughput and average response time, ρ, in a parallelised system can be stated as:

$$\rho = (s \cdot \mathit{simulq})/N_Q = \mathit{simulq}/\tau \tag{4.1}$$

The absolute throughput figure τ provides a good metric for the relative performance of different architectures with the same size of collection and number of machines. However, it is misleading to directly compare absolute throughput figures for different collection and cluster sizes. The fact that configuration A has higher throughput than configuration B tells us very little if configuration A has more machines, or a smaller collection: it is to be expected that throughput should increase in such a situation. The more important question is, has throughput increased by as much as might be hoped, given the change in configuration? To make answering this question easier, and comparisons between different configuration more straightforward, the concept of *normalised throughput* or $\bar{\tau}$ is introduced. Normalised throughput is absolute throughput adjusted for the size of the collection in terabytes, T, and the number of processing machines in the cluster, k, and can be stated as follows:

$$\bar{\tau} = (T \cdot \tau)/k = (T \cdot q)/(s \cdot k) \tag{4.2}$$

This expression has units of "Terabyte queries per machine second," and represents a collection-size and cluster-size independent measurement of throughput. Combining Equations 4.1 and 4.2, average response time can be derived from normalised throughput as follows:

$$\rho = (t \cdot k)/(T \cdot \bar{\tau}) \tag{4.3}$$

The focus of the research in this thesis is upon maximising normalised throughput. Maximal throughput is achieved by increasing the number of *simulq* until throughput plateaus; at this point, parallelisation benefits start to be outweighed by excessive load upon the system. Of course, workloads on a real-world system fluctuate over time; what is being tested here is the peak throughput that the system is able to handle. The precise number of *simulq* that is required to achieve maximum throughput for the experimental configuration is empirically determined in Chapter 5.

Given the trade-off between throughput and response time formulated in Equation 4.1, a statement such as "document partitioning achieves faster query response time than pipelining" requires qualification. In a fixed *simulq* environment, if system A has a higher throughput than system B for a given configuration and *simulq*, then system A also necessarily has a lower average query response time than system B for that configuration and *simulq*. It is possible that at a lower *simulq*, system B may have a higher throughput (and therefore lower average response time) than A. And it is certainly true that, in a serial or single-*simulq* environment, document partitioning's natural parallelisation will provide a faster query response time that pipelining's inherent serialisation. However, talking about query response times in isolation is misleading; it is also necessary to specify what level of query throughput the system is supporting. This trade-off between response time and throughput is revisited in Chapter 8.

4.3 Rank-biased dissimilarity

Efficiency-based IR research is not directly concerned with retrieval effectiveness. Indirectly, however, it is of great importance to verify that changes introduced for the sake of efficiency do not reduce retrieval effectiveness, or if they do, that the reduction is recognised, quantified, and justified as acceptable for the increase in efficiency gained. After all, if the quality of results is altogether ignored, an arbitrarily fast query evaluation system can easily be created. (Indeed, the author has inadvertently implemented a number of such systems in the course of this project.) Assessments of effectiveness therefore act as an important verification tool upon the retrieval correctness of efficiency-oriented techniques.

As described in Chapter 2, retrieval effectiveness is normally assessed by using a metric such as MAP to compare document rankings returned for a given collection and query set against a standard set of relevance judgments. There are times, however, when it is necessary to validate efficiency-oriented modifications against a collection or query set for which relevance judgments do not exist. In such cases, effectiveness assessments can be augmented by the use of a *dissimilarity metric*. The idea of a dissimilarity metric is to measure the degree of difference between two sets of ranked results. For verification of efficiency changes, this means measuring the difference between results obtained before and after the changes were made. If a difference is found, the dissimilarity metric cannot say whether the difference constitutes an improvement or a deterioration. Nevertheless, even if we assume that all differences involve a degradation in retrieval effectiveness, the dissimilarity metric provides an upper bound on the degree of the degradation.

A good dissimilarity metric has to meet a number of criteria. First, it should weight large changes in the ranking of any particular document more heavily than small changes. Second, because high ranked documents are of more interest to the user than low ranked ones, a good dissimilarity metric should weight changes towards the top of the rankings more heavily than changes towards the bottom, a requirement which other commonly-used measures such as Kendall's tau [Kendall, 1938] and Spearman's rank correlation coefficient [Moroney, 1951] do not satisfy. And third, a good dissimilarity metric should handle documents that occur in one ranking but not the other in a reasonable way. Note that it is not a requirement that a dissimilarity metric should take notice of the precise scores given to each document.

The dissimilarity metric used in the current research is defined as follows. Let $R(r)$ be the set of documents in the first r positions of some ranking R; and let $\ell(R, r, x)$ be the ranked position in ranking R of the document x, or $r + 1$ if

$x \notin R(r)$. The dissimilarity between two rankings R_1 and R_2 is calculated as follows:

$$\mathit{dissim}(R_1, R_2, r) = \sum_{x \in R_1(r) \cup R_2(r)} \left| \frac{1}{T + \ell(R_1, r, x)} - \frac{1}{T + \ell(R_2, r, x)} \right|$$

where $T \geq 0$ is a constant that determines the weighting between differences at the top of the ranking and those at the end of it. In this thesis, T is set to π (a value both reasonable and whimsically attractive), and r, the depth to which results are compared, is set to 1,000. It is more convenient to work with a normalised version of this metric, where the dissimilarity is expressed as a fraction of the score that would be obtained from completely disjoint lists. Thus, a normalised *dissim* of 0.0 indicates that two rankings are identical, and of 1.0 that they are entirely different.

4.4 Scalability

One of the most important questions about any proposed distributed query evaluation architecture is how well it scales, particularly when the resources available for use in the experimental evaluation are only a fraction of those deployed in a commercial web-scale search engine. In order for results obtained in the moderately-sized system used here (eight machines and half a terabyte of data) to be extrapolated with any confidence to larger datasets and more powerful environments, it is essential that experiments are carefully designed to answer the question of scalability.

There are two main dimensions of scale for an information retrieval engine: the size of the collection being indexed; and the amount of computing resources being employed. (In a fixed *simulq* environment, a third dimension, the rate of query arrivals, is determined by these two; in Chapter 8, we will examine system behaviour under varying query arrival rates.) Therefore, scaling can occur in three different ways:

1. increasing the size of the collection, while holding the computing resources constant;
2. increasing the computing resources, while holding the size of the collection constant; and,
3. increasing both the computing resources, and the size of the collection, holding the ratio between them constant.

Experiments along each of these directions of scale address a different question about real-world usage. The first direction of scale addresses how each architecture handles increasing data load on fixed hardware. The second addresses how well each architecture is able to make use of additional hardware resources to boost performance. And the third addresses how likely it is that the results achieved for an experimental setup of R computer resources handling a collection of size S will scale to a real-world deployment of nR machines handling a collection of size nS. The last direction of scale is the most crucial.

The scaling of computer resources is potentially a multi-dimensional proposition. Main memory, CPU speed, disk speeds, and so forth can all be separately varied. To simplify this potentially unmanageable situation, the unit of computing resource is fixed as a "machine", and scaling computer resources means adding more (identical) machines to the cluster. Chapter 8 will examine the effect of scaling one particular aspect of hardware configuration, namely the amount of main memory available to the system.

Collection size is the other experimental component that can be scaled. Here, the temptation is to make use of whatever real world collections of different sizes are available. Doing so has the specious attraction of also running the experiments on different types of collections: but it is precisely because of these differences that the temptation must be avoided. In scaling the collection size, it is only the size that should be varied, and not the composition of the data. If the makeup of the collection also changes, then two variables have been introduced instead of one, and it becomes difficult to determine which of these variables is the cause of differences observed in results. In the experiments in this thesis, smaller collections are created by extracting fractions of the full GOV2 collection, with care being taken to select documents at intervals from the repository, rather than taking the whole fraction from a contiguous segment of documents.

4.5 Controlling disk access variability

The current research assumes an environment in which indexes are too large to fit into memory, and are substantially disk-resident and disk-accessed at run time. It is postulated that in a disk-dependent environment, the superior disk access characteristics of pipelining will give it a performance edge over document distribution. Because the question of disk access behaviour is so central to the current research, it is essential that variability in disk performance is minimised.

Given that the same hardware is used for each experiment, disk performance is affected by two main factors. The first is the location of data on disk, and the second is its fragmentation. Disk location refers to whether data is placed on disk blocks that are closer to the spindle or to the rim of the disk platter. Because the rotational speed of the disk is constant, and the number of blocks per circumference is higher towards the rim of the disk than it is toward the centre, it follows that rimwards blocks are read faster than spindlewards ones. On the systems used in the current research, the read speed ratio is around 7/4. Additionally, if a file is highly fragmented on disk, then read performance deteriorates, as the disk head has to seek to different physical locations to access sections of a file that are logically adjacent. With these two factors combined, there is a very large disk read performance difference between well placed and poorly placed data. Experiments revealed that a system answering queries on a monolithic index of the full experimental collection spent 25% of its time waiting for disk I/O to complete when the data was poorly located, but under 10% when the data was in a better configuration.

One method of controlling disk variability is to create large, empty files once at the start of the experimental run, and then write each set of data into these files. In UNIX terms, this means opening the files without setting the `O_TRUNC` flag. However, this method is not in itself adequate for a distributed system, as it does not ensure that files on different machines are similarly located. Note that it is not, in general, possible with modern file systems to explicitly specify which blocks a file should occupy.

A better method of controlling disk variability, and the one employed in this thesis, is to set aside a dedicated experimental partition on each machine. The dedicated partition should be only slightly larger than the largest data set that needs to be deployed. The partition should be located on the same physical blocks on each machine; fortunately, it is generally possible to specify which blocks a partition should occupy. To obtain maximum transfer rates, the blocks chosen should be the rimwards-most ones. This is the setup used for all the experiments described in this thesis. Additionally, when each set of files was installed, a check was made (using a filesystem-specific utility) to ensure that no large block gaps had been created during installation. If a gap larger than a given size (0.5 GB was the size selected) had been created, then the files were deleted and reinstalled.

4.6 Summary

This chapter has described the experimental techniques developed during the course of the research presented in this thesis. The importance of using a statistically appropriate query set in efficiency experiments has been described, and a method for synthetically generating such a query set by translation has been presented. The use of *simulq* as a tuneable experimental parameter allows for query parallelism, and with it system load, to be set at the level that achieves maximum throughput for each distributed architecture. In the presence of query parallelism, there is a trade-off between throughput and average response time; setting *simulq* to a level that maximises throughput necessarily has a cost in response time. The question of scalability is of particular importance to distributed search engines, and the scaling of collection and cluster size need to be explored both separately and in conjunction. A dissimilarity metric is presented, which is preferable to those commonly used in the literature in that it weights changes at the top of the ranking more heavily, thus reflecting user priorities; this dissimilarity metric can then be deployed to ensure that the quality of results is maintained while efficiency changes are made. And finally, the disk access characteristics of different architectures are crucial to this thesis, and so controlling disk performance variability is essential; the use of dedicated experimental partitions for this purpose has been proposed as a best-practice technique. These techniques provide a firm foundation for the experimental results presented in Chapter 6. However, before proceeding to these experiments, it is necessary to deal with the issue of accumulator handling in the pipelined system, which is the topic of Chapter 5.

Chapter 5

Accumulator compression and pruning

As described in Chapter 3, the pipelined architecture ships the state of each query's evaluation between the nodes of the cluster. The great bulk of this data is made up of accumulators holding partial document similarity scores. The current chapter explores methods for minimising the number of these accumulators, and the size they take up once serialised for shipping. The first section examines techniques for compressing the shipped accumulators. In the second section, a novel dynamic accumulator pruning method is described. As will be seen in Chapter 6, the accumulator pruning technique benefits the throughput of all architectures. Finally, the third section calculates an upper bound on the volume of data that the pipelined architecture needs to ship between nodes once these accumulator compression and pruning techniques have been applied.

5.1 Accumulator compression

The great bulk of the data shipped in query bundles in the pipelined system consists of the partially-evaluated accumulators. There is a cost here both in bandwidth used, and in processor time to serialise the accumulators for shipment and unserialise them on arrival. If growth in accumulators is not constrained, then the number of accumulators can reach the number of documents in the collection, and in practice will usually be a non-trivial fraction of this. The full GOV2 collection has some 25 million documents. If each accumulator is represented as a 4-byte document number and a 4-byte floating-point accumulated score, and if (say) around 7 million accumulators have to be shipped for the more expensive queries, then over 55 MB of data has to be encoded, shipped, and decoded with each query bundle. For a system that is aspiring to process as many as 100 queries per second (as shown later), this is an impossibly large amount of data to be handling. Existing query pruning schemes propose that the number of accumulators can be limited to between 1% to 5% of the number of documents in the collection. However, even this fraction involves a large amount of data transfer; and, as is discussed later, there are some problems with existing methods. Therefore, space and processing optimisations in the handling of accumulators in query bundles will be of great importance to the performance of the pipelined system.

An accumulator stores two items: a document number, and the partial similarity score for that document. The space taken up in query bundles by document numbers can be reduced by storing them as document gaps, and then encoding them. Since Zettair uses the vbyte encoding scheme for its inverted lists (see Section 2.3), it is convenient to also use such a scheme for the accumulator document numbers. Taking 100,000 accumulators as the target accumulator set size (for reasons that will be explained later) and a collection with 25 million documents, the average document gap is around 250, and experiments show that the average size of a vbyte-encoded difference is 1.6 bytes, compared to 4 bytes for the unencoded document numbers.

The space taken in query bundles by partial document scores can be reduced by quantising them to integer values of a fixed bit size. Quantisation is not a new idea in the field, and has previously been employed on document lengths [Moffat et al., 1994] and on pre-calculated $w_{d,t}$ values in impact-ordered inverted lists [Anh et al., 2001]. Quantisation works as follows. Given a set of numerical values that are to be quantised, let L be the lowest value in the set, and let U be the highest value. Thus, L to U defines the range of quantisation. Let k be the number of distinct values available after quantisation; for instance, if quantising to a single byte, $k = 2^8$. The range L to U is divided up into k sub-ranges or buckets; each original value is assigned the number of the bucket whose range it falls into as its quantised value. Bucket range sizes can be assigned in a number of ways, including uniformly (that is, each bucket's range is the same size), or geometrically, to allow for greater precision at either the high or the low end of the range. Uniform quantisation is the simplest, and is the approach pursued here. Using uniform quantisation, the quantised value $\bar{v}$ of v in the range $0 \ldots k-1$ can be found using the formula:

$$\bar{v} = \left\lfloor k \cdot \frac{v - L}{U - L + \epsilon} \right\rfloor \tag{5.1}$$

where ϵ is an arbitrarily small positive value, introduced to ensure that $\bar{v} = k-1$ when $v = U$.

Metric	Accumulator representation	
	Unquantised	Quantised
Mean Average Precision	0.2368	0.2369
Dissimilarity	–	0.0045

Table 5.1: The effect of uniform quantisation of accumulators shipped in query bundles in the pipelined system. Quantisation is to 256 distinct values (that is, the largest amount representable by a single byte). The collection is `GOV2`. *Effectiveness is measured by mean average precision, using* `TREC` *topics 701-750 and the corresponding* `12-Nov-04` *qrels. Dissimilarity is measured as the dissimilarity between the two runs, using the first 100 queries of* `SYNQ` *and the normalised rank-biased dissimilarity metric, calculated to depth 1,000 results. An accumulator limit of 100,000 (see later) was employed.*

Table 5.1 shows the results in effectiveness and dissimilarity of quantising accumulator scores down to 1 byte or 256 distinct values before shipping them in the query bundle. This might seem a drastic loss of granularity, but (perhaps surprisingly) it has no cost in effectiveness. Since mean average precision is based upon a binary notion of relevance, one might suspect that the metric could be hiding reorderings of binary-relevant documents. However, the dissimilarity result dispels this notion: it confirms that not only does quantisation have little impact on effectiveness, it also makes very little difference to the ordering of results.

Quantisation and dequantisation can be implemented efficiently. The calculation of uniform quantisation is computationally simple, as can be seen from the formula itself. Dequantisation is not simply the inverse of the quantisation function, due to the downwards-truncation of quantised values. Instead, the dequantised value v' of bucket $\bar{v}$ is taken as the mean of the inverse quantisation of that and the next buckets' values, using the formula:

$$v' = \frac{(2\bar{v}+1)(U-L+\epsilon)}{2k} + L \tag{5.2}$$

But since only a small number of distinct integer values are involved, it is highly efficient to pre-calculate all $k = 256$ dequantised values for a given accumulator set's U to L range and keep them in a lookup table.

Taking document gaps and using a vbyte encoding scheme reduces the amount of space required for each document number to an average of around 1.6 bytes, and the quantised similarity scores require only a single byte. These techniques therefore reduce the size of each serialised accumulator from 8 bytes to an average of 2.6. Section 5.2 will examine methods of reducing the total number of accumulators that need to be shipped.

5.2 A new approach to accumulator pruning

In Chapter 2, the concept of query or accumulator pruning was introduced, and the two best-documented query pruning methods, the *Quit* and *Continue* schemes, were discussed. Recall that both the *Quit* and *Continue* schemes enforce an accumulator limit by changing their processing mode when this limit is reached. Initially, in both schemes, every new document pointer creates a new accumulator (`OR`-mode). When the accumulator limit is reached, the *Quit* method stops processing pointers, whereas the *Continue* method switches to allowing the scores for existing document accumulators to be updated, but not creating any new accumulators (`AND`-mode). The question remains of precisely when the accumulator limit is checked and the switch in processing modes is made. One option is that the switch be made only at the end of processing a term's inverted list, which will be termed the *Full* variant. Alternatively, the switch in processing modes could be made as soon as the accumulator limit is reached, even if this is in the middle of a term's inverted list; this will be termed the *Part* variant. With these variants, there are four different possible query pruning schemes: *Quit-Full*, *Quit-Part*, *Continue-Full*, and *Continue-Part*.

Each of the *Full* and *Part* variants has its potential problems. The *Part* variant strictly enforces the accumulator limit, but leads to an evaluation bias. For the term in whose inverted list the mode switch occurs, pointers occurring before the switch will be able to create new accumulators, whereas those occurring after it will not. Documents occurring earlier in the collection are therefore favoured over documents that occur later; and since the ordering of documents is arbitrary with regards to relevance, the retrieval effectiveness of the system is degraded. In the extreme, for a single-term query where the number of documents in which that term occurs is greater than the accumulator limit, later documents will not appear in the results at all, no matter how high their similarity score would have been under an exhaustive evaluation.

The *Full* variant avoids the inequitable treatment of documents occurring in different sections of the collection by deferring the processing mode switch to the end of the inverted list during which the accumulator limit was reached. However, from the point of hitting the accumulator limit up until the end of the list, new accumulators can still be created. This means that in the *Full* variant, the accumulator limit can and generally will be exceeded or *bursted*, often by a large amount. The problem is exacerbated if the terms in the query are high f_t ones, or alternatively in short queries where a term with an f_t below the accumulator limit is followed by a high f_t one. These types of queries are more common in the web-style queries that are the focus of contemporary research than in the longer, more discriminating traditional IR queries used in Moffat and

Zobel [1996]. Obviously, the bursting of the accumulator limit is particularly undesirable in a pipelined system.

Both the degraded effectiveness caused by the *Part* variant of the *Continue* strategy, and the bursting of the accumulator limit in the *Full* variant, were discovered in the course of the experiments carried out in this project, and led to the development of a greatly improved, adaptive query pruning scheme.

5.2.1 Adaptive pruning

There are three criteria that a good accumulator pruning scheme should meet. The first is that it should avoid bursting the accumulator limit. The second is that it should treat each document in the collection equitably. And the third is that accumulators, as a scarce resource, should be apportioned at any particular time to the documents that seem most likely to make the final ranked results. We have seen that the *Full* variant of the *Quit* and *Continue* schemes fails the first criterion, and the *Part* variant fails the second. The third criterion is more difficult to assess, but reflection suggests that to meet this criterion, a system should be able to remove an accumulator from an existing, low-scoring document and give it to a new, high-scoring one. It is true that, since terms are processed in reverse f_t order, and since f_t is inversely related to term weight, typically pointers in earlier lists will have more weight than pointers in later ones. However, there will be exceptions: a single occurrence of a low-f_t term in document d_1 may be outweighed by multiple occurrences of higher-f_t terms in document d_2, and in this instance it would be desirable (assuming the accumulator limit had already been reached) to remove d_1's accumulator and allocate it to d_2 instead.

Of course, the reassignment of accumulators from existing, low-scoring documents to new, high-scoring ones is not difficult to perform if efficiency is not an issue. One could, for instance, keep the accumulators in a heap, in addition to a lookup or merge structure, and always compare candidate new accumulators against the lowest-scoring existing one. However, the processing of each list would change from $O(n)$ to $O(n \log n)$ complexity. Indeed, even amongst linear-time operations, inverted list processing is such a central part of query evaluation, and involves such a large amount of data, that computational simplicity and efficiency is imperative.

A new query processing regime is proposed that meets the aforementioned three criteria (limit enforcement, equity, and effective allocation), as well as being computationally efficient. In the new regime, a partial document score threshold v is maintained. As the inverted list I_t for each term t is processed in merge-wise fashion against the existing accumulators list A, new accumulators

Algorithm 2 : Processing ranked queries

Input: a set of query terms t, their collection frequencies f_t, their inverted lists I_t, an accumulator limit L, and a function *contrib* which calculates a similarity contribution from f_t and $f_{d.t}$.

```
1:  assign $A \leftarrow \{\}$
2:  for each term $t$, in increasing order of $f_t$ do
3:      assign $v$ as described in Algorithm 3 below.
4:      for each document $d$ in $A \cup I_t$ do
5:          if $d \in I_t$ then
6:              assign $c \leftarrow contrib(f_t, f_{d,t})$.
7:          else
8:              assign $c \leftarrow 0$
9:          if $d \in A$ then
10:             assign $c \leftarrow A_d + c$
11:         if $c \geq v$ then
12:             assign $A_d \leftarrow c$ and $A \leftarrow A \cup \{A_d\}$
13:         else if $d \in A$ then
14:             assign $A \leftarrow A - \{A_d\}$
15:         pause periodically to reevaluate $v$, as described in Algorithm 3 below.
```

Output: a set of approximately L accumulator values, not yet normalised by document length

are created only when their initial score is greater than or equal to v. At the same time, existing accumulators whose score is below v are removed. The threshold v can therefore be used as a dynamically tuneable parameter to help the system maintain its accumulator limit L. If it looks like the limit will be exceeded, v is adjusted up, to decrease the number of new accumulators being created and increase the number of poorly-scoring existing ones being deleted; if, on the other hand, it looks like the limit will be undershot, v is adjusted down, with reverse effect. The process is summarised in Algorithm 2.

Still to be specified is how to choose v at the start of each term's inverted list, and how and when to adjust it during list processing (lines 3 and 15 of Algorithm 2). The choice of the initial value of v for each term's inverted list is made as follows. At the start of query evaluation, v is set to 0, and every new document occurrence creates an accumulator. Unconstrained accumulator creation continues until the next term to be processed could take the number of accumulators over the limit (that is, $|A| + f_t > L$). At the start of processing of the limit-threatening term t, an initial value for v is estimated as described below, and regular adjustments to v begin. From then on, v is carried forward from the end of processing one term to the start of processing the next.

The estimation of v at the start of processing of the limit-threatening term t is performed as follows. Assume that different $f_{d,t}$ values are spread uniformly throughout I_t. Assume also that there is some value h_t for which

Algorithm 3 : Adaptively estimate the thresholding parameter for a term.

Input: a set of accumulators A, a term t, an accumulator target L, and a contribution metric *contrib*. (Adapted from Lester et al. [2005])

```
assign $startA \leftarrow |A|$
if $|A| + f_t < L$ then
    assign $v \leftarrow 0$
else if $v = 0$ then
    assign $v \leftarrow contrib(f_t, \max\{f_{d,t} \mid d \in \text{the first } p \text{ pointers in } I_t\})$
assign $v_s \leftarrow v/2$
while pointers remain in $I_t$ do
    process pointers through until the $p$th as described in Algorithm 2, using
      $v$ as an accumulator value threshold.
    assign $predict \leftarrow |A| + (f_t - p) \times (|A| - startA)/p$
    if $predict > L \cdot \theta$ then
        assign $v \leftarrow v + v_s$
    else if $predict < L/\theta$ then
        assign $v \leftarrow v - v_s$
    assign $p \leftarrow 2p + 1$
    assign $v_s \leftarrow (v_s + 1)/2$
```

$|\{d \in I_t \mid f_{d,t} \geq h_t\}| = L$. Exactly one such document d could be expected to occur in the first $p = f_t/L$ pointers of I_t (where f_t is the total number of documents that t occurs in); therefore $f_{d,t}$ sets a lower bound on h_t, and so is an appropriate value to derive v from. So, the first p pointers of I_t are sampled, and the highest $f_{d,t}$ value in them is used to determine an initial v. Obviously, this is only an estimate, as the assumption of a strictly even distribution of $f_{d,t}$ values does not hold with real data, and the presence of existing accumulators, and the likely proportion of them to fail the threshold, have not been considered. Nevertheless, it provides a reasonable initial estimation.

The system re-estimates v at regular intervals. It does so by taking the change in the number of accumulators (both new accumulators created and existing ones dropped) since the start of processing the current list and extrapolating the change to the end of the list. If the extrapolation forecasts that the accumulator limit will be exceeded, v is adjusted up; if undershot, v is adjusted down. However, if the system adjusts v by too much, and particularly if it does so after it is well into processing the list, then the principle of equity will have been violated. At the same time, re-estimating v adds to the cost of the computation, and it is undesirable to have to repeat the re-estimation too often.

A number of techniques are employed to balance the requirements to treat documents equitably and to maintain the accumulator limit, while at the same time keeping computation costs down. First, the accumulator limit is given a tolerance, θ. The value chosen for θ depends upon the similarity metric being used; $\theta = 1.2$ was chosen for the experiments reported in the next section, which employed a language model with Dirichlet smoothing. If extrapolation suggests that at the end of the current list, $L/\theta < |A| < L \cdot \theta$, then v is left unchanged. Second, the amount v_s that v can be adjusted by is set to $v/2$ at the start of each list, and then is halved each time an adjustment is considered. The result is that large shifts are allowed at the beginning of the list, but they become increasingly small as more of the list is processed. And third, the initial adjustment point is after f_t/L pointers (the same as the size of the sample used to set the initial value of v), and each subsequent adjustment point is twice as far into the list as the one before it. This means that adjustments are made less frequently the further into the list processing has been performed, with no adjustment being made in as much as the last half of the list. The end result is a convergence to a stable threshold value, one that would ideally come about naturally, but is forced by the algorithm even if it does not. The treatment of documents at the very start of the list is less predictable than those later in the list, something that is unavoidable in an adaptive, single-pass scheme. However, the unpredictability is not biased (that is, does not systematically overestimate or underestimate the threshold), and the duration of the unpredictability is kept small. The process is summarised in Algorithm 3.

5.2.2 Evaluation methodology and results

Reducing the number of accumulators that need to be created, stored, and processed during query evaluation greatly benefits efficiency. The efficiency improvement occurs in all architectures, including the monolithic one. For the pipelined architecture, there is the added benefit of a decrease in the amount of data to be serialised, shipped between nodes, and deserialised. For all architectures, therefore, but particularly for the pipelined one, processing efficiency is an inverse function of the number of accumulators used, and the question becomes how low the accumulator limit can be set without significantly harming retrieval effectiveness.

The evaluation reported here uses the 426 GB `GOV2` collection, `TREC` queries 701-750, and the corresponding `12-Nov-04` qrels; the effectiveness metric is MAP. Evaluation is carried out using a monolithic Zettair system. Accumulators are counted by sampling accumulator usage after every 100 document pointers have been processed, over the whole of the query run, and averaging these counts. Regular sampling is preferable to counting accumulators in use at the end of each query or query term, as that understates the total system workload (accumulators by time) imposed by terms and queries that take longer to process.

Three query pruning regimes are chosen for evaluation: *Continue-Part*,

Target ('000)	*Continue-Part*		*Continue-Full*		*Adaptive-Pruning*	
	Actual ('000)	MAP	Actual ('000)	MAP	Actual ('000)	MAP
1	1.0	0.045	237.9	0.235	1.5	0.150
2	2.0	0.065	238.9	0.235	3.2	0.179
4	4.0	0.093	252.4	0.235	5.8	0.202
10	10.0	0.126	260.3	0.235	13.0	0.215
20	19.9	0.142	372.1	0.235	26.5	0.228
40	39.8	0.141	478.7	0.235	47.7	0.233
100	98.5	0.170	533.7	0.235	121.1	0.237
200	194.1	0.194	599.9	0.237	214.7	0.239
400	373.8	0.212	1,590.6	0.239	395.5	0.240
1,000	845.3	0.221	2,862.0	0.240	900.0	0.240

Table 5.2: Retrieval effectiveness scores, using TREC *topics* 701–750 *(short queries), the GOV2 collection, retrieval depth* $r = 1{,}000$*, and a language model with Dirichlet smoothing [Zhai and Lafferty, 2004]. Numbers reported are mean average precision (MAP), and the time-averaged number of accumulators required to process the query stream. A full evaluation of each query leads to a MAP of* 0.240*.*

Continue-Full, and *Adaptive-Pruning*. Results for the two corresponding *Quit* methods are not provided: while faster than *Continue*, they demonstrated much lower effectiveness. The MAP for a full evaluation of each query, with no limit on accumulators, is 0.240; the intent of the experiment is to see how few accumulators each method can use and still achieve a MAP that is not significantly lower than this.

Table 5.2 gives the results of the experiment. As expected, the *Continue-Part* strategy very strictly enforces the accumulator limit. However, its retrieval effectiveness is poor; even with an accumulator limit of 1,000,000 (4% of the 25 million documents in the collection), its MAP is significantly lower than that of a full evaluation, due to its bias towards documents occurring early in the collection. In contrast *Continue-Full* achieves effectiveness almost equivalent to that of full evaluation even with a nominal limit of 1,000 accumulators. However, it is very poor at enforcing the accumulator limit, and in practice cannot keep the number of accumulators below 1% of the collection size. Indeed, the limit set is an extremely poor indication of the number of accumulators actually created: attempting to explicitly set the limit to 1% of the collection size leads to use of accumulators numbering closer to 3% of the collection size. Note that for the *Continue-Full* method the accumulator limit of 1,000 roughly amounts to saying "except for very rare terms, only allow the lowest-f_t term to create accumulators"; it is interesting (and perhaps somewhat surprising) that relatively good effectiveness should result from such a rule.

In contrast to the *Continue-Part* and *Continue-Full* strategies, the *Adaptive-Pruning* method performs well both in enforcement of the limit and in retrieval effectiveness. For limits below 1% of the collection size, it does tend to overshoot the limit somewhat, sometimes by more even than the 20% tolerance allowed in the implementation of Algorithm 3. But significant overshooting only occurs for very low limits. The adaptive query pruning approach achieves full-evaluation effectiveness with 400,000-odd accumulators (or roughly 1.5% of the collection size), and near-equivalent effectiveness with a limit of 100,000 accumulators and actual accumulators in use numbering only 0.5% of collection size. Even below this accumulator limit, degradation in retrieval effectiveness is gradual. Moreover, in calculating MAP to a depth of 1,000 documents, we are being conservative, since few users actually look so deep into the ranking. With less conservative assumptions, even more aggressive accumulator pruning would be possible.

5.3 Calculating accumulator costs

Using the *Adaptive-Pruning* strategy, near-full retrieval effectiveness can be achieved with an accumulator limit L of around 0.4% of collection size. For the full GOV2 collection, this means in practice that an average of around 100,000 accumulators are used to process each query. Through the gap-coding of document numbers and the quantisation of partial document scores, the size of each accumulator in its serialised form can be reduced to an average of 2.6 bytes. Assume that the least frequent term t in each query has an $f_t \geq L$, so that the accumulator limit is reached by the end of processing of the first query term. The space each query bundle requires for accumulators is then roughly 260 KB.

The SYNQ query set used for the main experiments in this thesis has an average of 2.6 terms per query. Assume that every query term occurs in the index, and that for every query no two query terms are held on the same node. Then the number of times that accumulators need to be shipped for a query is one less than the number of terms in the query, since accumulators do not need to be shipped to the first evaluator node, or from the last one. Therefore, for the SYNQ query log, accumulators need to be shipped between nodes an average of 1.6 times per query.

Combining the averages for the bundled accumulator size and number of times accumulators need to be shipped, we arrive at an average bandwidth requirement of 420 KB per query processed in a pipelined system, even on the basis of the worst-case assumptions made in the above discussion. Given a 1 gigabit network, over 200 queries per second could be accommodated before network bandwidth became a bottleneck. As we shall see, such a rate of queries is well above the actual throughput any of the architectures tested here can achieve. For the pipelined architecture, therefore, the bandwidth required to ship query bundles is not a bottleneck, once accumulator compression and pruning techniques have been applied.

5.4 Alternative query modes

The preceding discussion of accumulator management in a pipelined system assumes that free-text queries are being evaluated using only term frequency information, which is the mode of querying specified in Section 2.1 and used throughout the current research. However, it is worth considering briefly how other query modes could be supported in a pipelined system. Boolean, phrase, and proximity queries, whether explicitly submitted by the user or implicitly generated by the search engine, require conjunctive relations to be evaluated. In a pipelined system, conjunctive processing can be performed by shipping the lowest f_t term t_1 to the node holding the second lowest f_t term t_2, then shipping the conjunction $t_1 \wedge t_2$ to the next lowest f_t term t_3, and so forth. The processing mode is therefore similar to pipelining accumulators. Where the conjunctive relation evaluated is a form of intersection (Boolean AND, required phrase, or required proximity), the size of the shipped conjunction is bounded by the frequency of the least-frequent term and is non-increasing (and in general sharply decreasing) as each subsequent term is processed. On the other hand, where the relation evaluated is a form of union (Boolean OR) or is being used as a supplemental evidence of relevance (proximity relations in addition to standard *tf·idf*), then in a naïve implementation the size of the shipped conjunction increases over time and is bounded only by sum of the frequencies of each term save the most frequent (processing finishing when the final node is reached). Even here, though, it should be possible to prune low-quality candidate documents, albeit not as aggressively as with simple *tf·idf* evaluation. These issues do not arise in a document-distributed system.

Contemporary web search engines also make use of query-independent evidence such as PageRank [Brin and Page, 1998] and click data [Xue et al., 2004]. Frequently, such evidence is attached not to terms, but to documents. In a term-partitioned index, such document-specific information could be replicated on each node, leading to increased memory usage. How significant the extra memory requirement is depends on the amount of information held in connection with each document, and also whether information is stored for every document or only for certain high-value ones. Alternatively, document-specific information could be held on a single node, which incorporates it in a post-processing step; however, such an approach would hamper the degree of accumulator pruning that could take place, as a document with a low accumulator value from query term processing could see its score increased by the application of document-specific evidence. In a document-distributed system, in contrast, storage and processing of per-document information is straightforward.

5.5 Summary

Efficient accumulator handling is vital to the efficiency of the pipelined architecture. This chapter has described techniques that greatly reduce the burden of shipping accumulators between pipelined nodes. With compression, each accumulator can be reduced to 2.6 bytes in serialised form. Dynamic accumulator pruning limits the number of accumulators created to around 0.4% of the number of documents in the collection with little or no cost in effectiveness. Combined, these techniques reduce the bandwidth required for a pipelined system to well within the capacity of a 1 gigabit network. In Chapter 6, it will be observed that the throughput of the pipelined architecture is only slightly more sensitive to changes in the accumulator limit than that of document distribution, indicating how effectively the techniques described in this chapter lighten pipelining's burden of shipping processing state between nodes.

Chapter 6

Experimental comparison of distributed architectures

Chapter 4 presented a methodology for performing efficiency experiments upon distributed information retrieval systems. Pipelining's potential bottleneck with accumulator shipment having been solved by the techniques presented in Chapter 5, we are now ready in the current chapter to employ the methodology to compare the document distributed, term distributed, and pipelined architectures. The first section introduces the data, explains the methods for testing scalability, and describes the experimental procedure. Suitable settings for *simulq* and the accumulator limit are determined empirically in the second section. The third section presents the experimental results for each of the three architectures, which demonstrate that pipelining scales poorly. Finally, the fourth section conducts a post-mortem on the experiments, to determine the reasons for pipelining's poor scalability.

6.1 Experimental data and procedure

The test collection chosen for the experiments in this chapter is the largest of the `TREC` collections to date—the `GOV2` collection, described in Section 2.4. Although the largest of the `TREC` collections, `GOV2` is by no means large by the standards of web-scale search engines. Nevertheless, there are a number of advantages to using it instead of a larger, but non-standard, crawl. First, the collection is readily available to other researchers. Second, the nature and contents of the collection have received some study, and existing performance results are known. And third, standard queries and accompanying relevance judgments are available, in the form of `TREC` topics 701-750 and the `12-Nov-04` qrels respectively, allowing

Attribute	Collection						
	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 (10^6)	0.39	0.79	1.57	3.16	6.31	12.60	25.21
Terms (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

Table 6.1: The various sample document collections. "Terms" is the number of distinct terms in the vocabulary. 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.*

the retrieval effectiveness implications of efficiency-oriented changes to easily be measured.

The query set used for the efficiency runs is the synthetically-generated SYNQ set, derived by translation from the Excite97 query log as described in Section 4.1.2. Prior to translation, the Excite97 log was processed as follows. First, the log was sorted into query arrival order, according to the timestamps provided. Then, Boolean and phrase queries were converted into simple ranked queries. Queries with terms longer than 15 characters, or with terms that looked like URLs, were removed. The terms "the", "and", "a", "are", "or", and "is" were stopped from the query log (but not from the index). An additional 310 common terms were stopped, but only if this did not reduce the length of a query to below 3 terms. Removing stop words from the query stream in this way reflects the common behaviour of real-world search engines. For the experiments reported in the current chapter, the first 20,000 queries were used; in Chapter 7, some experiments use the first 60,000 queries.

For these experiments the largest collection used was the full GOV2. Power-of-two fractions of this full collection were then extracted, as described in Section 4.4, down to a 1/64th collection. A collection that is $1/d$th of the full collection size is referred to here as a TB/d collection. The characteristics of each fractional collection are summarised in Table 6.1. Note that the collection scales uniformly in size, in number of documents, and in query cost. It is also of interest to observe the rate of growth in the size of the vocabulary: while sub-linear, a very substantial proportion of new terms are discovered each time the collection size is doubled. As a result, there are not great savings in the size of the on-disk vocabulary structure. However, a preliminary experiment shows that nearly all of the additional vocabulary will not be accessed during query processing; thus, the accessed vocabulary of TB/64 will be much the same size as that of TB/01.

Document partitioning was performed by dividing each fractional collection into equally-sized subcollections, taking care to select documents for each subcollection at intervals, rather than taking all documents for a subcollection from a contiguous segment of the original collection. The HTML documents in the original collection are stored in crawl order, meaning that higher-level web pages tend to cluster towards the front. Additionally, all the PDF files appear at the end of the collection. The round-robin selection mechanism ensures a higher degree of homogeneity between the document-partitioned subcollections. A hash function was used to assign terms to partitions in the term-partitioned indexes.

The hardware used in these experiments 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 via a 1 gigabit network, and controlled 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. Clusters of 2, 4, and 8 nodes, plus controller, are created from the full cluster. The subindex held by each node is stored on that node's local disk.

Each of the three distribution methods are tested against all combinations of the seven collections and three cluster sizes, excluding only those combinations for which the effective per-node collection is less than 1/64th of the full collection. In addition, a monolithic run is made on each collection, employing a single cluster node, to provide a baseline to compare the distributed results against. These runs enable the exploration of each of the three types of scale discussed in Section 4.4: scaling the collection size, scaling the cluster size, and scaling both together.

The experiments are designed to recreate as closely as possible the conditions that a real-world search engine would be running under. A live search engine would be making full use of main memory to buffer index data. Similar conditions need to be created for the experimental runs. Running the query set through twice, and timing the second run, is not an appropriate way to achieve this, even though it is a method that is sometimes employed. If, for instance, the data used in processing the query set is smaller in size than main memory, but the entire index is larger, then the system will be unrealistically caching exactly the section of the index that it needs to process the particular set of queries. On the other hand, running a warm-up query set drawn from an earlier section of the query log is by itself inadequate if the index is smaller than main memory; the whole index could be cached, but only those parts accessed in processing the warm-up query set actually will be.

To simulate a live system in the middle of an extended run, therefore, the following procedure is followed. First, indexes are copied from the cluster con-

Simultaneous	Method		
queries, m	Document	Term	Pipelined
1	2.05	0.54	0.60
2	3.57	1.04	1.13
4	4.48	1.20	2.01
8	5.21	1.23	3.01
16	5.77	1.24	3.94
32	6.03	1.22	4.46
64	6.21	1.21	4.50
96	6.18	-	4.40

Table 6.2: Normalised throughput with simulq m, using $k = 8$ nodes, and collection `TB/01`*. Throughput is presented in units of terabyte queries per machine second, as discussed in Section 4.2.1. "-" indicates that the run was unstable at this level of simulq.*

troller to the dedicated experimental partitions of the respective nodes. Then, the file cache is flushed by using the `mmap` system call to read through an unrelated file larger than main memory. Next, as much of the index as will fit into memory is cached using `mmap`, starting with the index vocabulary. When this is complete, the first 10,000 queries of the query set are run, but no timings are made. Then, the second 10,000 queries are run, and this run is timed. Finally, once the timed run is complete, the system is shut down, and the local indexes are erased.

6.2 Experimental parameters

Two experimental parameters introduced in Chapter 4 are the number of simultaneous queries or *simulq* allowed in the system at one time, and the accumulator limit set for the adaptive query pruning regime. For *simulq*, what is being sought is the setting that maximises system throughput. For accummulators, the appropriate limit for the full collection is already known, and the question is what limit to use for fractional collections and partitioned indexes. The appropriate settings for these parameters are determined empirically in this section.

6.2.1 Setting *simulq*

Table 6.2 shows the throughput for different *simulq*, for an 8-way partitioning of the full collection, using each of the three distribution methods. These figures emphasise the importance of parallelism in all query processing regimes, but particularly in the pipelined system. The optimal *simulq* setting for document distribution achieves a threefold improvement in throughput over a single-*simulq* or serialised system. For pipelining, the increase is more than sevenfold. Pipelining and document distribution both peak at 64 *simulq*, whereas term distribution reaches its maximum throughput at 16 *simulq*, and becomes unstable at 96, due to excessive load on the receptionist. It is possible that different collection and cluster sizes would display slightly different peak parallelism levels, but sample comparative runs indicate that 64 *simulq* is a reasonable level for all systems, and for the sake of comparability, this has been chosen as the level of parallelism for all configurations. A real-world system would not want to run for extended periods of time at this level of parallelism, as reponse times would necessarily be impaired. However, as discussed previously, our intent here is to determine maximum throughput.

6.2.2 Scaling the accumulator limit

In Section 5.2.1, it was demonstrated that with adaptive pruning, a limit of 100,000 accumulators is sufficient to attain retrieval effectiveness for the `GOV2` collection not significantly below that of full evaluation. The base accumulator limit for these experiments is therefore 100,000. However, two questions remain. Should the limit be scaled with the size of the collection? And in a document-distributed system, should each node have the full limit, or a fraction of it?

There are arguments for and against scaling the accumulator limit with collection size. On the one hand, if the size of the collection is halved, then roughly speaking so is each term's inverted list, and so too is the number of accumulators that would be created for each query were no limit set. On the other hand, the number of results to be returned to the user remains fixed at 1,000, regardless of collection size. Ultimately, the answer is best found experimentally: can the accumulator limit be scaled without significantly harming retrieval effectiveness, or is it necessary to maintain it at 100,000?

Table 6.3 gives the dissimilarities of scaling and not scaling the accumulator limit with collection size, compared to a full evaluation. As discussed in Section 5.2.1, the 100,000 limit does not give precisely the same effectiveness as no limit at all (the MAP is 0.237, rather than 0.240), which is reflected in the dissimilarity of 0.022 for the full `TB/01` collection. As one would expect, if accumulators are not scaled with collection size, then the dissimilarity between runs with and without an accumulator limit decreases as the collection gets smaller. On the other hand, if accumulators are scaled, then the dissimilarity grows as the collection size decreases, which is to say that the accumulator-limited results diverge farther from the accumulator-unlimited ones. This is the effect of maintaining the number of results returned to the user (and considered

Fraction f	Collection	Accumulators	
		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

Table 6.3: Normalised dissimilarity over 10,000 SYNQ *queries on a monolithic system, first holding the accumulator limit constant at* 100,000*, and then scaling the limit with collection size. Dissimilarity values are relative to a baseline of unlimited accumulators. For example,* TB/04 *is processed with* 100,000 *and then* 25,000 *accumulators. The dissimilarity metric is described in Section 4.3.*

for dissimilarity) at 1,000. For TB/64, for instance, the accumulator limit itself is only 1,666. Nevertheless, the increase in dissimilarity is only gradual, and even at its maximum is still relatively small. Therefore, in these experiments, accumulators will be scaled strictly with collection size. In a real system, one might choose instead to scale, but sub-linearly; however, this would needlessly complicate the analysis of the experimental results reported here.

A similar logic applies to the question of whether each node in a document-distributed system should have the full accumulator limit or a fraction of it. Recall that each of the k subindexes of a document-partitioned index is effectively an independent monolithic index of $1/k$th of the full collection. Since it has been determined that the accumulator limit L should scale with collection size, it follows that each node in a k-node document partitioned system should have an accumulator limit of L/k. The same is not the case, however, for term distribution or pipelining. In term distribution, query processing is performed on the receptionist, which will naturally use the full accumulator limit appropriate for the collection size. In the pipelined system, the processing state is shipped between nodes; each node therefore acts as if it were evaluating against the full collection, but only for its terms; therefore, each pipelined node should use the full accumulator limit for the collection.

6.3 Experimental results

6.3.1 Monolithic baseline

The efficiency experiments start with a run against each of the fractional collections indexed monolithically, to provide a baseline against which the distributed

Collection	Normalised throughput
TB/64	3.22
TB/32	4.44
TB/16	5.21
TB/08	6.16
TB/04	6.60
TB/02	6.66
TB/01	6.63

Table 6.4: Throughput as a function of collection size, using a single processor, with at most $t = 64$ queries concurrently active. Throughput is presented in units of terabyte queries per machine second, as described in Section 4.2.1.

architectures can be measured. The results are given in Table 6.4. Normalised throughput increases with collection size up until `TB/04` is reached, indicating that the system is benefitting from economies of scale. However, beyond that point, normalised throughput is flat, as the system becomes disk-bound. This suggests that a collection of roughly the size of `TB/02` is optimal for a machine with capacity of one of the cluster nodes. The likely effect of holding a significantly larger index on each cluster node will be considered in more detail in Section 8.2.

6.3.2 Document distribution

Table 6.5 is the first of three giving normalised throughput results for each of the three distribution methods, in this case for document distribution. These tables have a common format, with collection size scaling from left to right, cluster size scaling from top to bottom, and both scaling together from top-left diagonally to bottom-right. The top-left to bottom-right diagonal, for instance

k	Collection						
	TB/64	TB/32	TB/16	TB/08	TB/04	TB/02	TB/01
1	3.22	4.44	5.21	6.17	6.60	6.66	6.63
2	–	3.19	4.38	5.27	6.19	6.49	6.72
4	–	–	3.18	4.39	5.26	6.19	6.65
8	–	–	–	3.16	4.33	5.31	6.21

Table 6.5: Normalised 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 $t = 64$ queries were concurrently active. The receptionist is not counted as one of the processors.

k	Collection						
	TB/64	TB/32	TB/16	TB/08	TB/04	TB/02	TB/01
1	–	–	–	–	–	–	–
2	–	2.38	3.14	3.82	4.30	4.70	4.82
4	–	–	1.61	1.92	2.18	2.32	2.42
8	–	–	–	0.96	1.08	1.16	1.21

Table 6.6: Normalised 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.

from single-node `TB/08` to 8-node `TB/01`, will be referred to as the diagonal of *system scale*, and it indicates the ability of an architecture to handle larger collections by increasing the number of servers in the cluster while still maintaining throughput. The top line of Table 6.5 repeats the results for the monolithic run, for easy comparison.

As mentioned before, the index of each node in a document-distributed system processes queries very much as if it were an autonomous monolithic instance. In addition, every index in a k-node partitioning of a $1/f$ fractional collection is similar in size to a monolithic index of the $1/(f \cdot k)$ collection. Therefore, one would expect throughput figures along each top-left to bottom-right diagonal of system scale to be similar, which is indeed what is observed in Table 6.5. Note that the monolithic index of `TB/01` has similar throughput to the 2-node and 4-node partitions, indicating that for this index size and hardware, replication would be a reasonable strategy.

These results indicate that document distribution scales evenly with the size of the system as a whole (that is, when collection and cluster size are scaled together). Provided care is taken to avoid excessive network contention or overloading of the receptionist, and provided too that the homogeneity of partial collections is maintained, the diagonals of system scale can be confidently extended outwards to the bottom-right. The same is far from the case with the two term-partitioned architectures.

6.3.3 Term distribution

Table 6.6 gives the normalised throughput of the term distributed system along the various dimensions of scale. Monolithic results have not been reproduced here, as a one-node term-distributed system is not equivalent to a monolithic one (and would not make much sense in its own right). The immediate observation here is the term-distributed system's poor scaling with cluster size. Doubling

k	Collection						
	TB/64	TB/32	TB/16	TB/08	TB/04	TB/02	TB/01
1	3.22	4.44	5.21	6.17	6.60	6.66	6.63
2	–	3.79	4.63	5.49	6.11	6.47	6.81
4	–	–	4.05	4.81	5.38	5.62	5.98
8	–	–	–	3.60	4.02	4.31	4.50

Table 6.7: Normalised 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 $t = 64$ queries were concurrently active. The receptionist is not counted as one of the processors.

the number of nodes in the cluster halves the normalised throughput, meaning that the number of queries being processed each second remains the same.

What is occurring in Table 6.6 is that the receptionist is acting as a bottleneck upon the system as a whole. A little reflection will serve to demonstrate the reason. In the term-distributed system, each node is responsible only for looking up the location of a term's inverted list on disk, loading it from this location, and shipping it back to the receptionist. The nodes are, in essence, little more than inverted list servers. All actual processing of inverted lists is performed by the receptionist. As it happens, the receptionist machine in the experimental cluster has roughly twice the capacity of each evaluator node, and as a result for clusters of size $k = 2$, term distribution manages around 75% of document distribution's throughput. However, at this cluster size, the receptionist is already fully loaded, which is why normalised throughput halves when the cluster size is doubled.

Even these figures exaggerate the term-distributed architecture's processing capacity. When counting the number of machines k in the cluster in order to normalise the throughput figures, the receptionist is not included. For the document-distributed and pipelined systems, this is a reasonable approach, as the receptionist does very little work, and could if necessary share a machine with one of the evaluators. However, in the term-distributed system, the receptionist's processing resources are fully used. Therefore, we are essentially undercounting the machines in the term-distributed clusters by 2.

6.3.4 Pipelined

Table 6.7 gives the normalised throughput figures for the pipelined architecture. A comparison with the results for term distribution given in Table 6.6 demonstrates that pipelining is certainly a more efficient and scalable architecture to run on top of a term-partitioned index. For pipelining, the computation

Accumulators	Method		
	Document	Term	Pipelined
40000	7.26	1.43	5.73
100000	6.21	1.21	4.50
200000	4.91	1.02	3.57
400000	3.82	0.80	2.57

Table 6.8: Normalised 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 $t = 64$ queries were concurrently active. The receptionist is not counted as one of the processors.*

involved in query evaluation is shared amongst the cluster nodes, not concentrated on the receptionist, avoiding the bottleneck. Bandwidth usage is also lower. Through accumulator compression and adaptive pruning techniques, an upper-bound average bandwidth of 480 KB per query was calculated for TB/01 in Section 5.3, compared to the 2.02 MB needed per query for a term-partitioned index. The actual network usage as measured in the pipelined run was far less even than the upper-bound average, at 170 KB per query. A 1 gigabit network could therefore support around 600 queries per second, whereas the throughput of 4.50 terabyte queries per machine second for an 8-node index of TB/01 equates to around 85 queries per second. Clearly, bandwidth is not an issue for the pipelined system.

Compared to the even scaling of the document-distributed system, however, pipelining scales poorly. For smaller collections, pipelining outperforms document distribution, due probably to fewer network messages and vocabulary lookups, which remain fixed relative to collection size, and thus weigh more heavily with smaller collections. Pipelining is also comparable with document-distribution for larger collections on two-node systems. However, as the number of nodes in the cluster increases, pipelining's throughput drops off sharply. The difference is particularly stark along the diagonal of system scale from monolithic TB/08 to 8-node TB/01. Whereas the monolithic throughput of 6.17 terabyte queries per machine second is matched by 8-node document-distribution's 6.21, for pipelining, throughput has dropped to just 4.50. Pipelining, therefore, achieves reasonable results with small collections and small clusters, but scales poorly.

6.3.5 Adjusting the accumulator limit

Section 5.2, which discussed the adaptive pruning technique and the way it is used to maintain an accumulator limit, provided results showing that a limit of 100,000 is sufficient to achieve retrieval effectiveness that is not significantly worse than in an unconstrained system, when evaluating queries on the GOV2 collection. A limit of 100,000, therefore, has been used in the previously presented experiments, scaled with collection size as described in 6.2.2. Nevertheless, it is of interest to see the relative performance of different architectures if the accumulator limit is raised or lowered. For instance, an implementor may be unwilling to accept any loss in retrieval effectiveness, no matter how insignificant, and so may insist that the limit be raised to 200,000 or even 400,000. Conversely, the implementor may be willing to trade off a slightly greater degradation in retrieval effectiveness for higher throughput, particularly if users are interested in only the top $r = 10$ or $r = 20$ results, rather than the much larger $r = 1{,}0000$ results returned in these experiments. The effect of changes in the accumulator limit upon the pipelined architecture, relative to that upon the other architectures, is of particular interest. Because pipelining has to encode, ship, and decode the accumulators, it would be expected to be relatively sensitive to changes in the limit.

Table 6.8 shows the effect on throughput of varying the accumulator limit for the three architectures considered here. An 8-way partitioning of the full TB/01 index is employed in each case. Increasing the accumulator limit has a major effect on all of the distributed architectures. The throughput of pipelining drops by 55% as the accumulator limit is increased from 40,000 to 400,000. However, the throughput of document distribution also falls by over 45%. More accumulators means more memory usage, particularly in a highly parallelised environment, and also more computation in the merging of the accumulator and inverted lists. These factors affect every architecture, and the results reported in Table 6.8 indicate that the additional burden of shipping larger accumulator bundles which pipelining alone must bear is of relatively small significance.

6.4 Post-mortem

The experimental results given in the preceding sections are clearly not encouraging for the pipelined architecture. The supposition was that the pipelined architecture would outperform document distribution due to fewer disk seeks and a smaller per-node vocabulary. If this were the case, the effect should be more noticeable as the degree of partitioning increases (holding the ratio of collection to cluster size constant). The actual results, however, show quite the opposite trend: the greater the number of partitions, the worse pipelining's normalised throughput. Either the disk access and memory usage advantages predicted for pipelining failed to eventuate, or else some other detrimental effect is outweighing them. This is the subject of the current section.

System	Document-distributed		Pipelined	
	Sectors read ($\times 10^5$)	Distinct reads ($\times 10^3$)	Sectors read ($\times 10^5$)	Distinct reads ($\times 10^3$)
1, TB/08	6.89	10.06	6.89	10.06
2, TB/04	6.08	9.64	6.67	5.18
4, TB/02	6.38	9.81	3.93	2.30
8, TB/01	5.79	8.79	3.32	1.64

Table 6.9: Disk read performances of the document-distributed and pipelined architectures along the main diagonal of system scale, from a monolithic index of TB/08 *to an 8-node index of* TB/01*. The results are averages per node. All figures refer to physical reads to disk (that is, they exclude accesses to file data cached in main memory), as reported by the* /proc *filesystem under Linux kernel 2.6. "Sectors read" is the number of sectors read off disk (a whole sector is read, even if only a single byte is required). "Distinct reads" refers to the distinct read requests made by the filesystem layer to the disk.*

Table 6.9 examines the read behaviour of the document-distributed and pipelined architectures along the most important diagonal of scale. As the system scales in size, each node in the document-distributed system continues to perform roughly the same number of read operations. In contrast, the number of read operations made by each node in the pipelined system falls dramatically as the system grows larger. Pipelining is benefitting from a number of confluent factors as the system scales up. First, the number of terms held on each node falls, requiring fewer distinct reads from disk. The greater length of inverted lists also means that, secondly, there is less unused data read off disk per sector, especially with rare terms, so the number of sectors read goes down. Thirdly, the vocabulary is smaller, so there is more memory free for caching files. And fourthly, the smaller number of query terms means fewer vocabulary reads. These figures demonstrate that the pipelined architecture does, indeed, achieve the improved disk read and memory use characteristics predicted for it.

Unfortunately, pipelining encounters a problem that far outweighs the benefits it derives from its parsimony in disk reads and memory use. The problem is its poor balancing of load between the nodes in the system. The poor load balancing of the pipelined system can be observed in Figure 6.1, which gives the average busy load per node for 8-node indexes of TB/01 under the document-distributed and pipelined architectures. Here, *busy load* refers to the amount of time a node's processor spends doing anything except the idle task; it includes time spent waiting for I/O to complete. The busy load is remarkably even between nodes in the document-distributed system, and each node is at around 99% load, indicating excellent utilisation of the cluster's processing capacity. In contrast, load is very poorly balanced between the nodes of the pipelined sys-

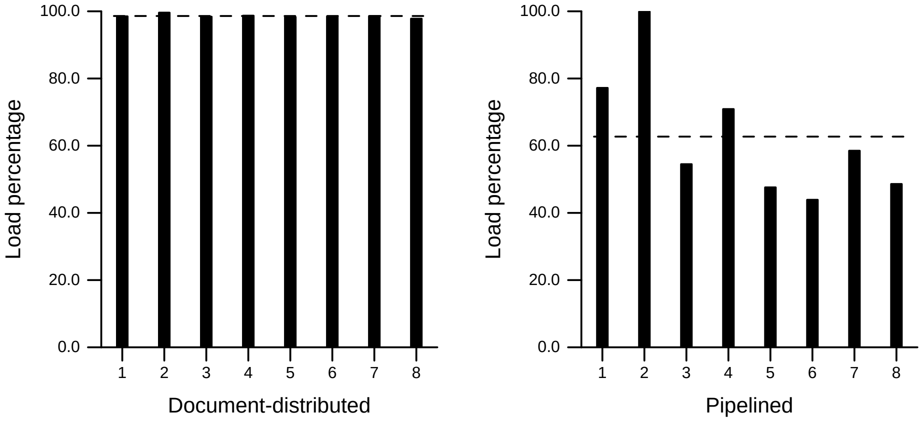


Figure 6.1: Average per-processor busy load for $k = 8$ and `TB/01`*, for document-distributed processing and pipelined processing. The dashed line in each graph is the average busy load over the eight processors.*

tem. One node is at 100% load, while the others languish between 40% and 80%. As a result, processing utilisation across the system does not even reach 65%, a very poor result. The underutilisation of processing capacity greatly outweighs the disk access and memory usage efficiencies of the pipelined system.

Figure 6.2 breaks down the busy load of each architecture over time. Here, too, we can observe the document-distributed nodes sharing the workload across the entire run, varying only slightly and then in unison, and never leaving a full processor's worth of processing capacity idle until the system winds down over the last few dozen queries. Next to this, the load allocation of the pipelined system appears chaotic, with first one, then another node becoming busy, while others are nearly completely idle for short periods, and the system as a whole always leaves at least two processors' worth of processing capacity unused. The only consistent feature of this graph is that node two is almost always at full capacity.

The nature of the problem with the pipelined system is clear. One node (here, node two) becomes fully loaded. Queries start queueing up for processing at this node, starving the other nodes of work. As a result, processing resources are underutilised, and the cluster as a whole slows down.

Pipelining's imbalance in processing load derives directly from an imbalance in the volume of index data that each node has to process. Figure 6.3 gives the total volume of inverted lists processed on each node over the 10,000-query run for both architectures. Data volume is shared between the nodes of the document-distributed system with almost perfect evenness, while pipelining is

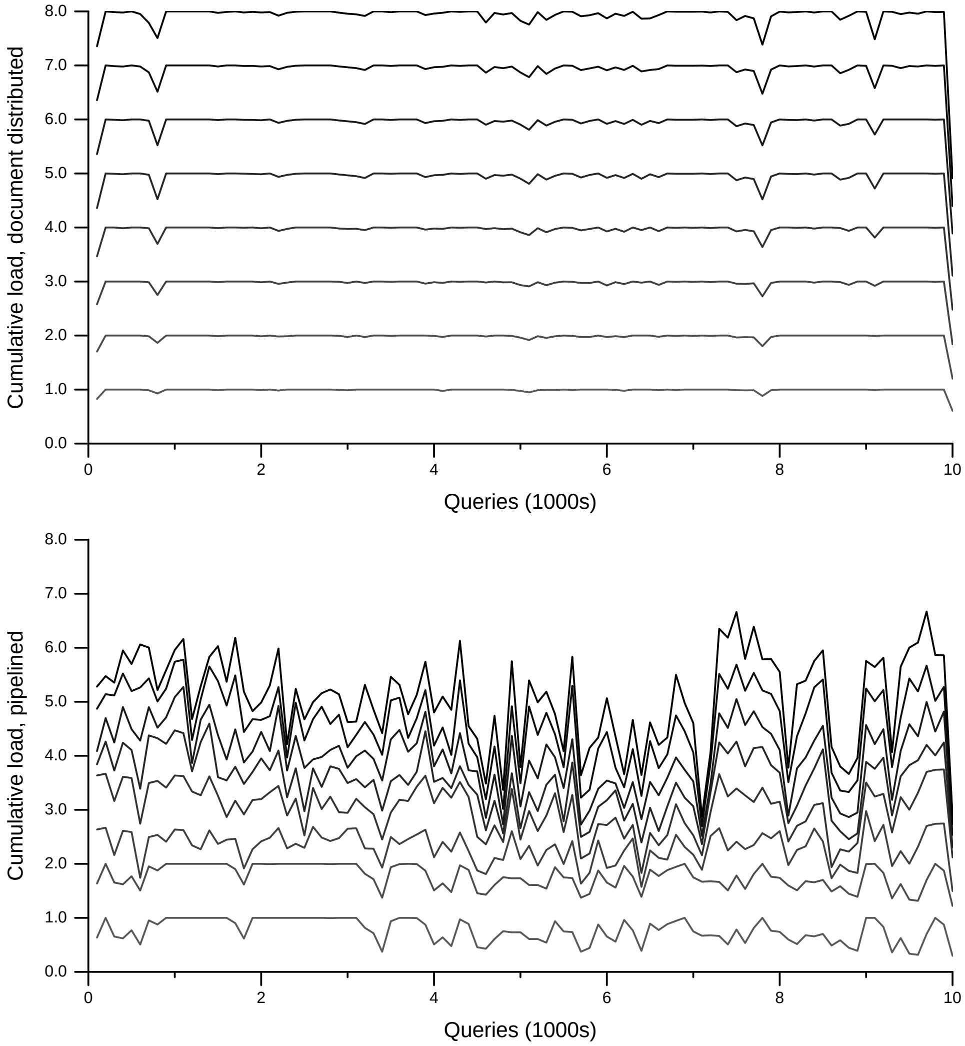


Figure 6.2: Aggregate load over the eight processors in a $k = 8$ run on `TB/01`*. The eight lines in the top graph reflect the pointwise aggregate load utilisation of the eight processors in a document-distributed system; the eight lines in the lower graph reflect the pointwise aggregate load utilisation of the eight processors in an equivalent pipelined system. Load is measured every* 100 *queries in a* 10,000 *query run.*

highly uneven, with node two having the lion's share of data to process. The correlation between Figures 6.3 and 6.1 is clear.

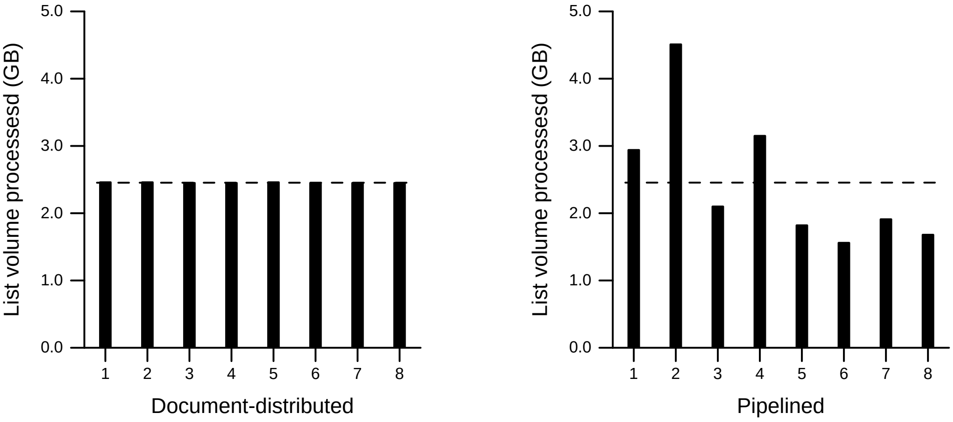


Figure 6.3: Total inverted list lengths processed on each node, in GB, for a $k = 8$-node run on `TB/01`*. Both systems process the same total volume of index data, but share the work differently.*

6.5 Summary

In this chapter, the document distributed, term distributed, and pipelined architectures have been subjected to a thorough empirical examination, using the experimental techniques established in Chapter 4. Document distribution has been shown to scale evenly with system size, whereas term distribution fails to scale at all. Pipelining achieves the disk access benefits predicted for it, and operates well within the bandwidth capacity of a gigabit network. However, uneven load balancing means that, rather than gaining from the greater contiguity of data that increases in system scale offer, it in fact scales poorly, and achieves significantly lower throughput than document distribution on an 8-node cluster. Chapter 7 examines the reasons for pipelining's poor load balancing, and explores possible solutions.

Chapter 7

Load balancing

In Chapter 6, it was observed that the pipelined architecture suffers from poor balancing of workload between evaluator nodes. As a result, pipelining's performance lags behind that of a document-distributed system. The current chapter examines the issue of load balancing in a pipelined system in some detail. The first section presents a simple model for measuring and simulating workload imbalance, and uses it to calculate the expected imbalance resulting from the arbitrary assignment of terms to nodes. Section 7.2 examines different methods for attempting to overcome workload imbalance by intelligent term assignment, estimates their effectiveness using the simulated model, and evaluates their performance experimentally. Finally, in the third section, load balancing through term replication is introduced, differing degrees of replication are examined, the different query routing strategies that replication enables are described, and these degrees of replication and routing strategies are assessed both in simulation and experimentally.

7.1 Statistics of query set imbalance

Section 4.1.1 introduced the concept of the workload L_t of term t, and defined it as the product of t's frequency in the query set, $\mathcal{Q}_t$, and the length of t's inverted list, B_t. From L_t can be derived the workload L_n of the evaluator node n in a pipelined system, as:

$$L_n = \sum_{t \in T_n} L_t$$

where T_n is the set of terms that are handled by node n. That is to say, the workload of a node for a given query set is the sum of the workloads of all terms held on the node, which is the total length in bytes of all inverted lists that the node will process during the evaluation of the query set. The node's simulated workload provides a useful estimation of the actual processing load that each node will be subject to during a run.

From the simulated workloads of the nodes in a pipelined system, the simulated *workload imbalance* of the cluster of nodes can be calculated. A cluster's workload imbalance is the ratio of the highest node workload to the mean workload of all nodes in the cluster. The ratio of highest to mean is a more pertinent measurement of workload evenness than standard deviation, since the results from the Chapter 6 indicate that it is the busiest node in the system that is liable to become the bottleneck for the system as a whole.

In the term-partitioned indexes examined in Chapter 6, terms were allocated to nodes using a hash function. Hashing provides an arbitrary but deterministic method of term allocation. For the experimental query set and collection, and the particular hash function used, hashed allocation produces a workload imbalance of 1.87. However, it is not correct to say that 1.87 is the workload imbalance of arbitrary allocation, since each different method will result in a different allocation of terms to nodes and hence a different workload imbalance. It is only possible to talk of the average or expected workload imbalance of arbitrary allocation. Generating a large number of (pseudo-)random allocations gives an average workload imbalance of 1.45. The hashing technique has been "unlucky" to produce an imbalance of 1.87, which is in the worst 2% of outcomes from random allocations. Nevertheless, even the average result of arbitrary allocation is a highly imbalanced system.

The poor average workload imbalance resulting from arbitrary allocation is due to term workload skew. The question of skew in the distribution of term workloads was previously examined in Table 4.2, where it was seen that the highest workload term, "free", by itself makes up roughly 6.2% of the workload of the query set, with "new" making up a further 2.8%. The effect that such skew has upon workload imbalance can be seen from a rough calculation. Assume that all terms apart from "free" and "new" have the same workload. Then imagine that all other terms are assigned evenly to nodes, with only these two highest workload terms remaining to be assigned. Each node would then hold roughly $91.0/8 = 11.4\%$ of the workload. Then assign the remaining two terms. If they go to different nodes, the node receiving "free" will have 17.6% of the workload, compared to an average of 12.5%, giving a workload imbalance of 1.41. If they both go the same node, that node will have 20.4% of the workload, giving a workload imbalance of 1.63. These figures are consistent with the average imbalance resulting from random allocation.

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
Initial L_t	1.14	1.20	1.20	1.37	1.30	1.24
Previous L_t	1.14	1.26	1.23	1.19	1.17	1.20

Table 7.1: Simulated load imbalances with different term assignment strategies. Each batch reflects a simulated evaluation of the workload of 10,000 *queries, across $k = 8$ processors, as a ratio of the largest workload to the average workload, for that strategy. The last column shows the average imbalance over the five query batches.*

7.2 Load balancing through term assignment

The obvious solution to arbitrary allocation's poor load balancing is to assign terms to nodes in a manner that attempts to balance workload. A prediction of future workload is made, and then terms are assigned so that each node's predicted workload is roughly similar. The simplest basis for predicting the workload of term t is the term's frequency in the collection or f_t, which is a factor in the term's workload and can be calculated at index construction time; however, this method does not account for a term's frequency in the query set. If a sample of the query set is available, the workload prediction can be based upon that observed in the sample. A single sample could be taken, and term allocation made semi-statically based upon it; this will be referred to as the "Initial L_t" method. Alternatively, terms could be dynamically reallocated at intervals, based upon the workloads observed in the immediately preceding interval; this is referred to as the "Previous L_t" method.

Once a prediction of each term's future workload has been made, terms are allocated to nodes using a fill-smallest algorithm. The fill-smallest algorithm works as follows. First, terms are ordered by decreasing predicted workload. Then, iteratively, the heaviest remaining workload term is allocated to the least loaded ("smallest") node, choosing an arbitrary node in the event of a tie. The process is repeated until all terms are allocated.

Table 7.1 gives the workload imbalances resulting from the different allocation strategies. The simulated runs have been extended to five query sets, being the 2nd through 6th 10,000 query batches of `SYNQ`. In each case, the preceding 10,000 batch is used by the "Past L_t" method to calculate historical load and dynamically re-allocate terms before each batch, whereas the "Initial L_t" allocates terms based upon batch 1 and uses this allocation for all batches; as a result these two methods have the same allocation for batch 2. No results are

Strategy	Batch					Avg
	2	3	4	5	6	
Hashed	4.50	4.54	5.04	4.80	4.59	4.69
Initial L_t	5.62	5.33	5.70	5.43	5.39	5.49
Doc-distributed	6.21	6.30	6.64	6.60	6.49	6.45

Table 7.2: 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 queries executed against the* `GOV2` *collection to identify the top $r = 20$ matching documents, and where the numbers reported are in units of terabyte queries per machine second. 64 simulq were used for each run. The last column shows the average throughput over the five query batches.*

provided for batch 1, as there is no prior sample for either the "Past L_t" or "Initial L_t" methods to use. These figures show that random allocation provides an expected imbalance in the 1.4 to 1.5 range. Using f_t as an estimator of workload achieves an average imbalance of 1.34, but is highly variable for each individual batch. Both the high average imbalance and the variability within this average are indicators that f_t is a poor estimator of L_t; the variability is due to the inherit "randomness" in the suitability of allocation decisions that results from f_t's inaccuracy as a predictor of workload (sometimes good choices happen to be made, sometimes poor ones). Previous workload with dynamic re-allocation is a much better and more reliable predictor. Taking a single sample and semi-statically allocating based on it performs almost as well, and avoids the need for dynamic re-allocation.

Table 7.2 shows the throughput resulting from implementing the "Initial L_t" allocation method in a pipelined system. Term allocation was performed statically, based upon the workload of the first 10,000 queries of the query set. Results are also given for pipelining with hashed allocation, and for document distribution, to facilitate comparison. Intelligent term allocation increases the throughput of the pipelined system some 20% over that of hashing; however, it still falls well short of the throughput of document distribution.

Strategy	Batch					Avg
	2	3	4	5	6	
Hashed	1.60	1.60	1.49	1.64	1.68	1.60
Initial L_t	1.18	1.21	1.13	1.40	1.40	1.27
Doc-distributed	1.01	1.01	1.01	1.02	1.01	1.01

Table 7.3: Observed busy load imbalance for the runs described in Table 7.2

Table 7.3 gives the observed busy load imbalances for these runs. Here, observed busy load is defined as the proportion of time each node spends doing anything except the idle task, and the imbalance is the ratio between the busiest node and the average for the cluster. These observed balances should be compared with the estimated ones from Table 7.1. The estimates for "Initial L_t" turn out to be quite good, both on average and for each data point. As predicted, intelligent term allocation only partially resolves the load imbalance of the pipelined system.

These results demonstrate that the allocation of terms to nodes based upon historical workload significantly improves the workload imbalance of the pipelined system, and as a result boosts throughput. However, pipelining still falls well short of the even workload balance of the document-distributed architecture, and therefore fails to achieve document distribution's throughput.

7.3 Load balancing through partial replication

This section examines the technique of partial replication as a way of overcoming the load balance problems that term assignment by itself is unable to fully resolve. In a partially-replicated pipelined system, certain terms are allocated to two or more evaluator nodes. This means that the workload for these terms can be shared between those nodes. In addition, the routing of queries through the system is no longer fixed. Instead, for queries that include replicated terms, the system has a choice of nodes to route the query bundle through. The choice can be used to respond dynamically to load imbalances as they occur at run time. Intelligent query routing offers the possibility not only of compensating for the imperfect estimation of future term workloads, but also of ameliorating localised workload imbalances.

Query terms should be chosen for replication so as to offer the maximum ability to smooth out workload while minimising the number of terms replicated and the total data size of the replicated information. Minimising replication is desirable not only because it reduces disk space usage (and, if replication is to take place dynamically, bandwidth usage and processing costs), but also because replication tends to lessen caching benefits. If a term is duplicated, then it requires twice as much memory for every instance of that term's information to be cached.

The obvious terms to replicate are high-workload ones. Selection of these can be performed using a query stream sampling method similar to that used in the workload-based term allocation strategies discussed above. For these simulations, the "Previous L_t" sampling method will be used. Note that it would be particularly inefficient to choose terms for replication based on f_t alone, without

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-1/10/100	1.05	1.12	1.09	1.16	1.12	1.11

Table 7.4: Simulated load imbalances with different amounts of index list replication. All assignment is via the fill-smallest term assignment strategy based on the "Previous L_t" approach. When multiple servers host a query term, queries are routed to them alternately. Each batch reflects a simulated evaluation of the workload of 10,000 *queries, across $k = 8$ processors, as a ratio of the largest workload to the average workload, for that strategy. The last column shows the average across all 5 batches.*

regard to $\mathcal{Q}_t$, as this would increase the size of the replicated data relative to the amount of redundant workload made available. A similar logic applies to the question of how often to replicate each term. If used, multiple replications should be skewed towards the highest workload terms. It is very possible, in particular, that it is more efficient to replicate the highest few workload terms multiple times than to provide a single replication of a greater number of lower workload terms.

Once the terms to replicate have been chosen, replicated and unreplicated terms have to be assigned to nodes. For the current simulations, a fill-smallest algorithm is used to allocate both replicated and unreplicated terms, with workload prediction again being based on the "Previous L_t" method. All instances of a higher (predicted) workload term are allocated before any instances of a lower one, and the assigned workload of each instance of a replicated term is divided by the number of replications; that is, if term t with total predicted workload L_t is replicated to R nodes, then the predicted workload of each node is increased by L_t/R for the purposes of the fill-smallest algorithm. The fill-smallest algorithm is modified to ensure that pairs of duplicated term instances do not always line up together.

Term replication makes query routing possible. The simplest routing strategy is to choose alternately between nodes. If high-workload terms are chosen for replication, alternating routing will help even out imbalances caused by term workload skew. Term t replicated c times effectively changes from a single term with workload L_t to c terms each with workload L_t/c. However, alternating routing foregoes the opportunity to compensate for imperfect workload prediction and localised imbalances at runtime.

Table 7.4 gives the simulated balance for the alternating routing strategy on an 8-node system. Batches 2 through 6 of the synthetic query log are assessed separately and averaged. For each batch, the $n \in \{1, 10, 100, 1000\}$ highest-workload terms of the preceding batch are replicated. One multiply-replicated system is also simulated; in this, eight-way replication is performed of the top workload term, four-way replication is performed to depth 10, and duplication is performed to depth 100. Both replicated and non-replicated terms are allocated to nodes using the fill-smallest algorithm. The baseline to measure these results against is the average imbalance of 1.20 achieved by the term-assigned but unreplicated "Previous L_t" method (see Table 7.1). The results in Table 7.4 indicate that only limited benefit is gained by replication when an alternating routing strategy is adopted. The prediction of future workload that historical workload provides is an imperfect one, and so assignment choices based upon it are themselves imperfect. Replicating terms reduces workload skew, which in turn helps to flatten out the unevenness caused by the difference between historical and current workload. However, replication by itself cannot fully balance workload.

Instead of simply alternating between choices, a routing strategy can be based upon keeping some measurement of the workload for each node. Then, routing decisions can be used to try to even the workload out. A simple routing strategy is for the receptionist to keep a count of the total workload sent so far to each node. The count is updated for each query term before the query it is in is dispatched. Then, if a particular query term is replicated on multiple nodes, the node that has the lowest workload count is chosen. This approach will be referred to as the *historical* routing strategy. It can be regarded as a version of the fill-smallest algorithm, extended temporally.

Table 7.5 provides the simulated workload balances attained by the historical routing scheme upon a pipelined system with differing degrees of replication. For

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-1/10/100	1.01	1.00	1.00	1.00	1.01	1.00

Table 7.5: Simulated load imbalances with different amounts of index list replication. All assignment is via the smallest-first term assignment strategy based on the "Previous L_t" approach (see Table 7.1). Historical workload is used to determine routing choices for queries that contain replicated terms.

replication and assignment purposes, workload is predicted by the "Previous L_t" method. These figures demonstrate that the use of intelligent routing on a replicated system is able to achieve near-perfect balance with a relatively small degree of replication. Specifically, duplicating the 100 highest workload terms brings the system close to perfect balance on all query batches; then, further either duplicating the next 900 highest workload terms, or multiply-replicating the first 10 (as described previously) achieves essentially perfect balance. By the use of intelligent routing, the system is able at run-time to remedy imbalances that occur due to the differences between historical and current term workload.

7.3.1 Routing schemes

Simulation has indicated that partial replication combined with intelligent routing aimed at balancing workload over time is able to achieve perfect simulated workload balance over the length of the 10,000 query set. However, a possible objection to the historical routing strategy is that, in determining the workload of each node, it places the same weight upon terms from queries that occurred thousands of queries ago as it does upon terms from queries sent out just recently. In other words, while it can achieve global balance, it could be argued that it is not sensitive enough to short-term imbalances.

One solution to the perceived insensitivity of the historical routing strategy to short-term imbalances is to decay the workload over time by some decay factor $0 < d < 1$. Before adding the workloads from terms in each query, existing node workload counts are multiplied by $1 - d$. This places more weight on recent queries than on remote ones. This will be referred to as the *decayed-historical* routing strategy.

Another way of calculating node workload is to only count terms from queries that are currently in process—recalling that the system processes several queries in parallel—and ignore prior workload altogether. When a query enters the system, each node's workload count is increased by the workload of terms in the query that it handles, as with the historical routing strategy. Then, when the query is finished, these workloads are subtracted from each node. This will be referred to as the *work-in-progress* routing strategy.

Although the work-in-progress routing strategy only counts queries that are currently in the system when calculating node workload, it still has an imperfect picture of node workload at any particular time. A query moves through the system a node at a time. If a query has already passed a node, its terms' workloads should not be counted in the node's current workload. In addition, if a query has yet to reach a node, it might be desirable to treat its workload differently from that of a query that is currently at that node. Similarly, whether a term is going to be processed at the start of the query's route or at the end might make a difference in how the load for each node is calculated when deciding the route for that query.

These problems with the timeliness of the information used in the work-in-progress routing strategy could be addressed by undertaking a more complex simulation of query traversal at the receptionist, to try to get a more accurate estimation of the actual load at a given evaluator node at the time that a particular query is estimated to arrive at that node. However, no matter how sophisticated the simulation is, it will always remain no more than an approximation. A simpler and more reliable method is not to decide upon the query route at the receptionist at all, but allow each evaluator node to decide upon the next hop for each query. Where the next term in a query is held by more than one node, the evaluator node that has just finished with the query (or the receptionist, if it is the first term in the query) asks each of the candidate nodes to report their current workload, and forwards the query to the node that has the lowest current workload. Here the workload is represented simply as the total length in bytes of all query terms currently in process. This will be referred to as the *evaluator-based* routing strategy.

The advantage of the evaluator-based routing strategy lies in the timeliness of the information it receives. There can still be a lag between when the workload data is received from candidate nodes and when the query actually arrives at the chosen one, but the inaccuracy is much less than could be achieved by simulation. On the other hand, there is a cost in the slight increase in latency, bandwidth, and processing involved in nodes fetching workload information over the network from candidate successor nodes. In addition, while each evaluator node knows more accurately than the receptionist what its current load is, the receptionist has a better idea what its future load is likely to be, in the form of queries in the system which have not yet reached that node; so in switching from receptionist-based routing to node-based routing, some information is lost. Finally, counting current load in the way described above is arguably too coarse: one would like to consider not how long each in-process inverted list is, but rather how much of it is left to process, although this would involve interrupting the accumulator-processing pipeline in each thread to negotiate mutex locks and update global counters.

Since the alternatives to the simple historical routing scheme aim at improved short-term load balance, it is not possible to assess their merits using our simple simulated model, which only accounts for workload globally across the query set. It is necessary therefore to turn to implementation to assess the merits of these different routing schemes.

Table 7.6 gives the results of using the different routing schemes described

Routing Strategy	Throughput	Busy load	Load balance
Historical	6.11	88.5%	1.08
Decayed-historical	6.05	87.4%	1.07
Work-in-progress	6.22	90.2%	1.06
Evaluator-routed	6.27	92.0%	1.06

Table 7.6: Throughput, busy loads, and observed load imbalances of different routing strategies, for the second 10,000 *queries of* `SYNQ`*, run on an 8-node index of the full* `GOV2` *collection. Throughput is measured in units of terabyte queries per machine second. Busy load is the average proportion of time each node spends doing anything except the idle task. Load balance is the busy load ratio between the busiest node and the average. The routed strategies are all based upon a term-partitioned index with the 100 highest-workload terms duplicated. Workload is estimated from the first* 10,000 *queries of* `SYNQ`*. Duplicated and unduplicated terms are assigned to nodes in a fill-smallest manner.*

above with batch 2 of the synthetic query log and an index with the top 100 workload terms duplicated. Workload is estimated based on that observed in batch 1 of the query set. The decayed-historical approach has a decay of 0.01, giving query term workloads a half-life of 69 queries ($0.99^{69} \approx 0.5$), which accords with the simultaneous query setting of 64, and indeed higher decays were found to give worse performance. Term replication provides significantly improved throughput for all the routing schemes considered here compared with the throughput of 5.62 terabyte queries per machine second achieved by term assignment (compare with Table 7.2), and the throughput differences between the routing schemes are only slight. Interestingly, the decayed-historical approach performs worse than the historical scheme. The historical approach is able to achieve long-term balance, which means that there is an implicit bias towards recent history; increasing the bias through decaying history actually harms throughput. The work-in-progress and evaluator-routed schemes do seem to have a slight performance edge. The busy load balance figures indicate that actual load is not balanced as evenly as the estimated model suggests it should be; this is because actual load is determined by other factors than gross inverted list length alone. System utilisation, as measured by average busy load, is lower than the load imbalance by itself accounts for, suggesting that short-term imbalances are occurring. Similarly, evaluator routing increases throughput and system utilisation compared with historical routing by more than it improves global load imbalance, suggesting that its superiority lies chiefly in ameliorating short-term imbalances. The question of short-term load imbalances will be looked at in more detail later.

Partitioning and replication method	Total index size (GBs)
Unreplicated term-partitioned	16.06
Duplicate 100	16.60
Multi-1/10/100	16.82
Multi-4/20/100	17.14
Duplicate 1000	18.21
Document-partitioned	18.35

Table 7.7: Total size in gigabytes of all indexes in an 8-way partitioning of a full index of `GOV2` *for different partitioning and replication strategies.*

7.3.2 Degree of replication

The results presented in Table 7.5 show that duplicating the top 100 terms achieves near-perfect simulated workload balance, and that either additionally multi-replicating the top 10 terms, or duplicating the top 1,000 terms, achieves perfect balance. Of course, duplication carries a cost, in higher disk space usage, dissipation of caching effectiveness, and (if dynamic list re-allocation is being performed) in shifting inverted lists between nodes. Additionally, simulated balance is not a perfect predictor of throughput results, especially where different degrees of replication are all reporting close-to-perfect balance. This section will examine the costs and experimental performance of different degrees of replication.

Table 7.7 outlines the cost in additional disk space of replicating a term-partitioned system. Costs involved in dynamically re-partitioning the index are not considered here, as a semi-static sampling and partitioning approach is assumed; however, were dynamic re-partitioning to be implemented, then its costs would be related to the amount of replicated data. Note, first, that document partitioning results in a significantly greater total index size (here, 14% larger) than term partitioning, as the vocabulary on each node is larger. There is, therefore, a lot of leeway to replicate terms in the term-partitioned index and still have a smaller total index size than under document partitioning. Even replicating the 1,000 highest workload terms still takes less room than a document-partitioned index. Table 7.5 indicated that duplication to depth 100 plus multiple replication of the top 10 terms offered equivalent balance under simulation to duplication to depth 1,000. From Table 7.7, it can be seen that multiple replication involves much less disk space usage than additional duplication, which suggests that it provides a better balance-to-space trade-off. Indeed, extending eight-way replication to depth 4 and four-way replication to depth 20 is still more space efficient than duplication to depth 1,000.

Table 7.8 shows the normalised throughput achieved in an actual run by dif-

Degree of replication	Normalised throughput
Duplicate 100	6.27
Multi-1/10/100	6.30
Multi-4/20/100	6.35
Duplicate 1000	6.38

Table 7.8: Normalised throughput of an evaluator-routed pipelined system with different degrees of replication, measured in units of terabyte queries per machine second. All runs were made with the second 10,000 *batch of the* `SYNQ` *query set upon an 8-node partitioning of* `TB/01`*.*

ferent degrees of replication. Greater replication does allow for higher throughput, although the differences are only slight. Note that whereas the simulation in Table 7.5 showed that the multi-1/10/100 configuration is able to achieve perfect simulated workload balance across the query set, the figures given here indicate that a higher degree of replication continues to benefit throughput, by giving the system more choices to route around short-term imbalances. Combined with the disk space usages reported in Table 7.7, these figures suggest that multiple replication of high-workload terms is more efficient than extending duplication to a depth of 1000.

Table 7.9 compares the evaluator-routed, partially-replicated pipelined system against hash-assigned, unreplicated pipelining and document distribution, across query batches 2 through 6 of `SYNQ`. Term workload estimation and assignment for the partially-replicated system is based on the workload observed in the first 10,000 query batch, rather than being reallocated for each batch based upon the immediately preceding one, in order to avoid the cost of the reallocation. With partial replication and intelligent routing, the pipelined system achieves a small but consistent edge in throughput over document distribution,

Strategy	Batch					Avg
	2	3	4	5	6	
Hashed	4.50	4.54	5.04	4.80	4.59	4.69
Multi-4-20-100	6.35	6.44	6.65	6.89	6.91	6.65
Doc-distributed	6.21	6.30	6.64	6.60	6.49	6.45

Table 7.9: Measured query throughput rates, in units of terabyte queries per machine second, on an 8-node partitioning of `TB/01`*, where each batch consists of* 10,000 *queries. 64 simulq were used for each run. The partially-replicated system replicates the top 4 workload terms 8 times, the next 16 top workload terms 4 times, and the following 80 top workload terms 2 times. Term workload is predicted based upon the "Initial L_t" method. The last column shows the average throughput over the five query batches.*

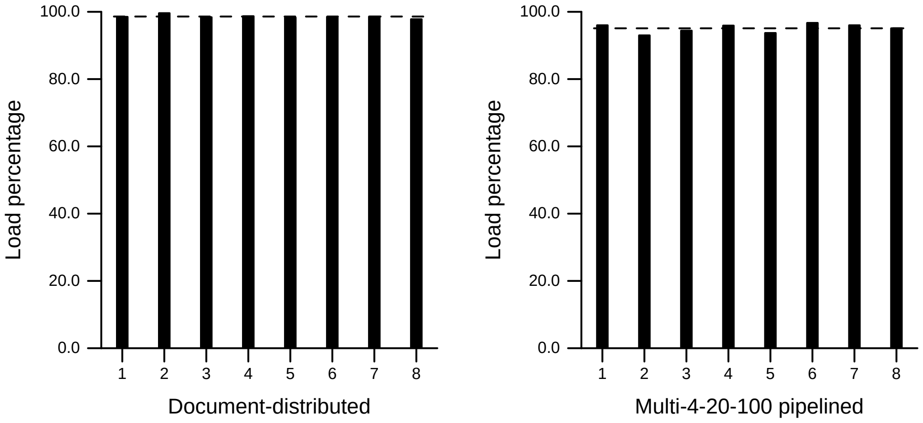


Figure 7.1: Average per-processor busy load for $k = 8$ and `TB/01`*, for document-distributed processing and partially-replicated pipelined processing, for the 2nd* 10,000 *of* `SYNQ`*. The partially-replicated system replicates the top 4 workload terms 8 times, the next 16 top workload terms 4 times, and the following 80 top workload terms 2 times. The dashed line in each graph is the average busy load over the eight processors. These results can be compared with Figure 6.1 on page 69, from which the document-distributed graph is repeated.*

and more than a 40% increase in throughput over an unreplicated pipelined system with hashed assignment.

Figure 7.1 compares the actual per-processor load of the partially-replicated evaluator-routed pipelined system against document distribution. As can be seen, the system achieves excellent busy load balance between the nodes; the load balance figure is 1.02. However, system utilisation at around 95% is still below the full utilisation achieved by document distribution.

To see why the partially-replicated pipelined system is still unable to achieve full system utilisation, Figure 7.2 examines what is happening with busy load at a finer granularity. The busy load of node 2 is plotted against the average busy load in the system at intervals of 100 queries. Note that system-wide busy load is fairly stable. However, node 2's busy load dips on a number of occasions. Most of these coincide with a rise in load imbalance and a fall in overall system utilisation, suggesting that a bottleneck is developing around one of the nodes in the system at these points. Although workload across the 10,000 query batch is fairly evenly divided, short-term imbalances persist, and even evaluator-based routing is unable to entirely resolve them.

A higher degree of replication gives the system more choices to route around localised imbalances. The slight improvements in throughput observed with greater replication in Table 7.8 point in this direction. It is possible that an

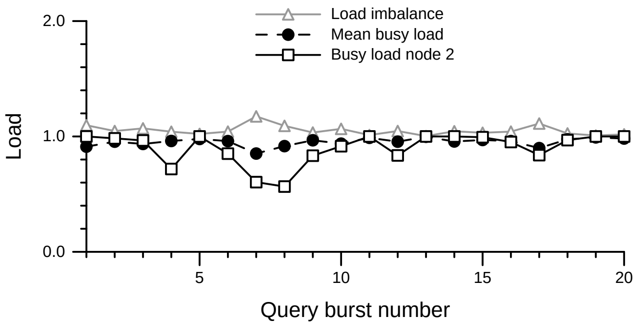


Figure 7.2: Busy load and load imbalance measure for 20 "bursts" of 100 queries. The busy load of node 2 is plotted against the average busy load of all 8 nodes, and against the load imbalance between busiest and average node.

improved routing scheme might be able to squeeze another small increment in throughput out of the additional replication. However, the small differences observed between existing routing schemes observed in Table 7.6 suggest that any benefit gained from fine-tuning the routing strategy would only be marginal. Furthermore, as the degree of replication is increased, cache effectiveness decreases, because information for a single term takes up space in the data caches of more than one node in the system.

7.4 Summary

This chapter has examined the issue of load balance in a pipelined system in detail. A simple model for approximating load balance has been introduced, and the model has been used to calculate the expected load balance of arbitrary allocation. Different methods of load balancing through intelligent term allocation have been examined. These have been shown to significantly improve upon the load balance of arbitrary allocation, and as a result to increase throughput by some 20%. However, term allocation alone still falls well short of the load balance and throughput achieved by document distribution. In response, the strategy of replicating high-workload terms on more than one node has been proposed. Using partial replication and intelligent routing, the pipelined system achieves higher throughput than document distribution, albeit only marginally. These results have examined only the configuration of an 8-node partitioning of the full TB/01 collection. In Chapter 8, the scalability of these results is examined. In addition, the document-distributed and pipelined architectures are tested under a more memory-limited environment, and the question of the relationship between throughput and average response time is revisited.

Chapter 8

Scalability

Chapter 7 investigated different techniques for load-balancing the pipelined architecture, and demonstrated that a partially-replicated pipelined system achieves marginally higher throughput than document distribution on an 8-node partitioning of TB/01. The current chapter investigates the scalability of these findings. The first section extends the timings along the three directions of scale that were discussed in Section 4.4, namely scaling by collection, cluster, and system size. In the second section, the effect of reducing the amount of memory available to each node in the system is examined, not only as a configuration option of interest in its own right, but also because it (imperfectly) approximates scaling the index size beyond that achievable with our sample collection. Finally, in the third section, the relationship between query arrival rate, average response time, and system throughput is revisited experimentally, and the implications for response times of increasing system scale are examined.

8.1 Scaling replication

Section 4.4 discussed the importance of testing the scalability of a distributed system in three different ways: scaling the size of the collection while holding the size of the cluster constant; scaling the size of the cluster while holding the size of the collection constant; and scaling the size of the collection and the size of the cluster at the same time. All three forms of scale were explored in the results given in Chapter 6. Chapter 7 evaluated different term allocation and replication strategies against a single data point, that of an 8-node partitioning of TB/01, finding that, with partial replication and intelligent routing, pipelining provides marginally higher overall query throughput for that particular data point. In this section, the benefits of replication are explored along the three directions of scale. Note that while the multiple replication strategy was found

k	Collection		
	TB/04	TB/02	TB/01
$k = 2$	6.14	–	7.01
$k = 4$	–	6.23	6.71
$k = 8$	5.25	5.85	6.27

Table 8.1: Normalised throughput, in units of terabyte queries per machine second, for evaluator-routed pipelined distributed retrieval, for the second 10,000 *queries of the* SYNQ *query log. The top 100 historical workload terms are duplicated, and all terms are assigned in a fill-smallest manner based on historical workload. Up to 64 queries are simultaneously active in the system at one time.*

in Chapter 7 to offer the best performance, mapping multiple replication onto 4- and 2-node systems is problematic, and for this reason, duplication to depth 100 is the mode of replication used in this chapter. Queries are routed using the evaluator-based routing strategy.

Table 8.1 gives the results of running the standard synthetic query set on different collection and cluster sizes. The data points along the directions of scale leading to the 8-node partitioning of TB/01 have been selected. The key feature of these figures is the results achieved along the diagonal of system scale, that is, the three data points consisting of the 2-node partitioning of TB/04, the 4-node partitioning of TB/02, and the 8-node partitioning of TB/01. Recall from Section 6.3.4 that the unreplicated pipelined system demonstrated very poor scalability along this diagonal, with normalised throughput figures of 6.11, 5.62, and 4.50 terabyte queries per machine second respectively. In contrast, with replication and intelligent routing, the normalised throughput of the pipelined system records a slight but perceptible increase along this diagonal—a significant achievement. A comparison with the results for the document-distributed architecture given in Table 6.5 on page 63 shows that the partially-replicated pipelined system outperforms at almost all data points, albeit only by a marginal amount for the larger collections.

8.2 Memory-limited operation

In the experiments presented so far in this thesis, processing resources have been scaled in units of identical machines. It is also interesting to consider the effect that scaling individual aspects of the hardware configuration has on the document-distributed and pipelined architectures. Perhaps the most pertinent modification to consider is a reduction in main memory. With less memory on each machine, a smaller proportion of the index can be cached, and so the system becomes more disk-dependent. An added experimental attraction of a

k	Collection		
	TB/04	TB/02	TB/01
$k = 2$	4.43 *−28%*	–	4.66 *−31%*
$k = 4$	–	4.41 *−29%*	4.56 *−31%*
$k = 8$	4.17 *−4%*	4.84 *−8%*	4.32 *−30%*

Table 8.2: Normalised throughput for document-distributed retrieval, for the second 10,000 *queries of the* SYNQ *query log, with available main memory on each processor node reduced from* 1024 MB *to* 512 MB. *Throughput is measured in units of terabyte queries per machine second. The top 100 historical workload terms are duplicated for the pipelined run. Up to 64 queries are simultaneously active in the system at one time. The percentage figures give the change in throughput compared to the full-memory results provided in Table 6.5 on page 63.*

more memory-limited environment is that it helps to compensate for the modest size of our test collection. Although it is the largest of the TREC collections available to date, in terms of web scale GOV2 is by no means large. As reported in Table 6.1, a monolithic index of the full collection is 16.6 GB in size. Spread across 8 servers each with 1 GB of main memory, it is obvious that a significant proportion of the index can be cached in memory. Limiting memory provides insight into the performance characteristics that could be expected from a larger experimental collection.

A reduction in available memory is most easily achieved by running in parallel a low-priority process that allocates and occupies exactly the amount of memory that is to be made unavailable. To force each page into physical memory, the process traverses repeatedly through its allocation, marking each page by writing a single byte to it. The page marking needs to be done frequently enough to prevent the occupied memory being swapped out, but not so frequently that a significant proportion of processor time is required. In the following runs, 512 MB of the 1 GB of main memory on each node is occupied by a background process, which performs a marking-traversal of this memory once every 500 milliseconds. When run on an otherwise idle system, the memory-occupying process took up less than 1.5% of processor time.

Table 8.2 reports the throughput achieved by the document-distributed architecture in the memory-limited environment, along with the change in throughput compared to the full-memory configuration. With the reduced amount of memory available for caching, the throughput of the document-distributed architecture falls by around 30% for configurations on or above the diagonal of system scale from 2-node TB/04 to 8-node TB/01, while 8-node TB/02 falls by 8% and 8-node TB/04 only marginally.

k	Collection		
	`TB/04`	`TB/02`	`TB/01`
$k = 2$	5.84 *−5%*	–	5.45 *−22%*
$k = 4$	–	6.25 *+0%*	6.39 *−5%*
$k = 8$	5.10 *−3%*	5.73 *−2%*	6.13 *−2%*

Table 8.3: Normalised throughput for partially-replicated, evaluator-routed pipelined distributed retrieval, for the second 10,000 *queries of the* `SYNQ` *query log, with available main memory on each processor node reduced from* 1024 `MB` *to* 512 `MB`. *Throughput is measured in units of terabyte queries per machine second. The top 100 historical workload terms are duplicated for the pipelined run. Up to 64 queries are simultaneously active in the system at one time. The percentage figures give the change in throughput compared to the full-memory results provided in Table 8.1.*

Table 8.3 gives the throughput figures for the partially-replicated pipelined architecture under the same memory-limited conditions. In contrast to document distribution, pipelining suffers only a slight fall in throughput for all indexes except 2-node `TB/01`. The results along the diagonal from 2-node `TB/04` to 8-node `TB/01` are particularly interesting. The throughput of 2-node `TB/04` falls by 5% with the reduction in main memory. However, ss the system scales along the diagonal, data contiguity increases, with the results that the 4-node `TB/02` system maintains the same throughput as in the full-memory environment despite the halving of main memory—a remarkable result. At the same time, though, there is a slight fall in the throughput of the 8-node `TB/01` configuration, suggesting that the decrease in main memory complicates the task of load balancing.

The disk access characteristics underlying the throughput figures from Ta-

Document-distributed				Dupl-100 pipelined			
k	Collection			k	Collection		
	`TB/04`	`TB/02`	`TB/01`		`TB/04`	`TB/02`	`TB/01`
$k = 2$	32.7	–	37.2	$k = 2$	5.6	–	28.2
$k = 4$	–	33.2	36.6	$k = 4$	–	1.7	4.1
$k = 8$	2.0	9.3	35.9	$k = 8$	1.2	1.6	2.7

Table 8.4: IO wait loads as percentages for document-distributed and partially-replicated, evaluator-routed pipelined distributed retrieval, for the second 10,000 *queries of the* `SYNQ` *query log, with available main memory reduced from 1GB to 512MB, as described in the text. The top 100 historical workload terms are duplicated for the pipelined run. Up to 64 queries are simultaneously active in the system at one time.*

bles 8.2 and 8.3 are shown in Table 8.4, which states, for each configuration, the average percentage of time the evaluator nodes spent waiting for I/O operations (almost exclusively disk reads) to complete. The pipelined architecture's superior disk access characteristics stand out clearly. In the reduced memory environment, both architectures are forced to read more data off disk for processing. Disk access becomes a bottleneck for the document-distributed system on the 2-node `TB/04` to 8-node `TB/01` diagonal, with the architecture's fragmented inverted lists leading to many short reads, and much time lost in seeking between disk locations. In contrast, the partially-replicated pipelined architecture, with its greater contiguity of data on disk and longer continuous reads and fewer seeks, has no trouble in keeping the processor supplied with data, and only begins to encounter serious delays for the 2-node `TB/02` configuration. In short, in this memory-limited environment, the document-distributed architecture becomes disk-bound with a per-server index a quarter of the size of that required to make the pipelined system similarly disk-bound. Looking in more detail, note that the I/O wait load for the pipelined system drops from 5.6% for 2-node `TB/04` to 1.7% for 4-node `TB/02`, demonstrating the on-disk data contiguity benefits that pipelining gains from increased system scale.

These results indicate, most obviously, that the search engine deployer is able to save money on memory by using a suitably balanced pipelined architecture, which is not an inconsiderable point in itself. More importantly, though, the results suggest that, were a larger collection to be hosted on each machine, or were a less compact index representation employed (for instance, one that includes word offsets), the reduction in the proportion of the index cached would be more deleterious to the document-distributed architecture than to pipelining. These results are of particular interest to truly web-scale indexes, which are several orders of magnitude larger than the ones used in the current experiments. The precise performance trade-offs will depend of course on hardware configuration, data size, and software implementation. Nevertheless, the superior performance of the pipelined architecture in a predominantly disk-bound environment is clear.

8.3 Throughput, response time, and system load

The experiments in this thesis so far have been focused upon maximising query throughput, and have used *simulq* (the number of queries simultaneously active in the system) as a parameter tuneable to achieve maximal throughput. The question of the system's average response time (the time from the moment a query is received to the moment it is answered) is also of real-world interest. This section examines the relationship between throughput and response time

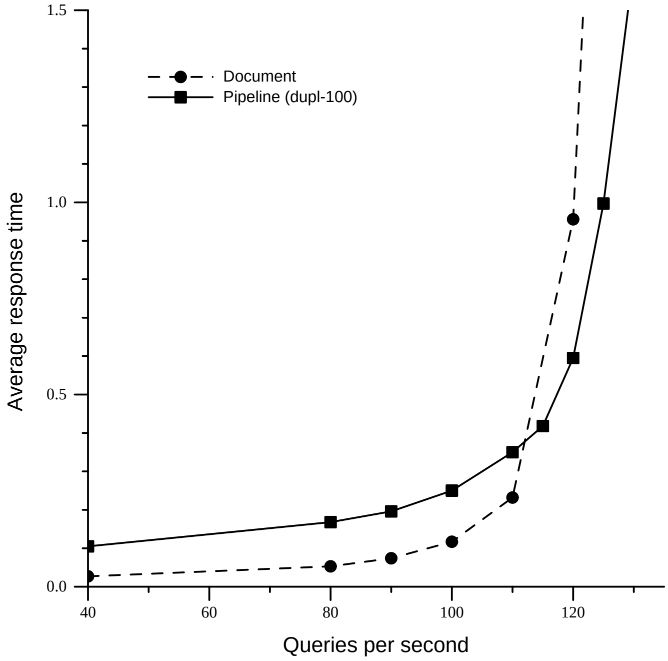


Figure 8.1: Average query response time for the document-distributed and pipelined systems. The pipelined system duplicates the 100 historically highest workload terms, and performs evaluator-based routing. Both runs were made on an 8-node cluster hosting an index of `TB/01`*. The queries were the 2nd* 10,000 *of the* `SYNQ` *query set, with following queries added to the queue until all* 10,000 *in the set had completed. Queries were released into the system queue at regular intervals.*

under varying degrees of system load.

The most natural experimental methodology for exploring questions of query response time and workload is to vary, not the number of queries simultaneously active, but the number of queries submitted to the system per second, with queries being released at regular intervals. Varying the interval between query releases approximates how workload is conceived of in real-world settings, as the number of queries that arrive to be processed in a given unit of time. Of course, in a real system, query arrivals would not be evenly spaced, with something like a Poisson distribution being more likely, but this simplification is adequate for the current purposes.

Figure 8.1 displays the average response times achieved by the document-distributed and partially-replicated pipelined architectures for different rates of query arrivals per second, for an 8-node partitioning of `TB/01`. At low workloads, the pipelined system takes almost four times as long to answer each query as the document-distributed system does. The difference is due to the natural

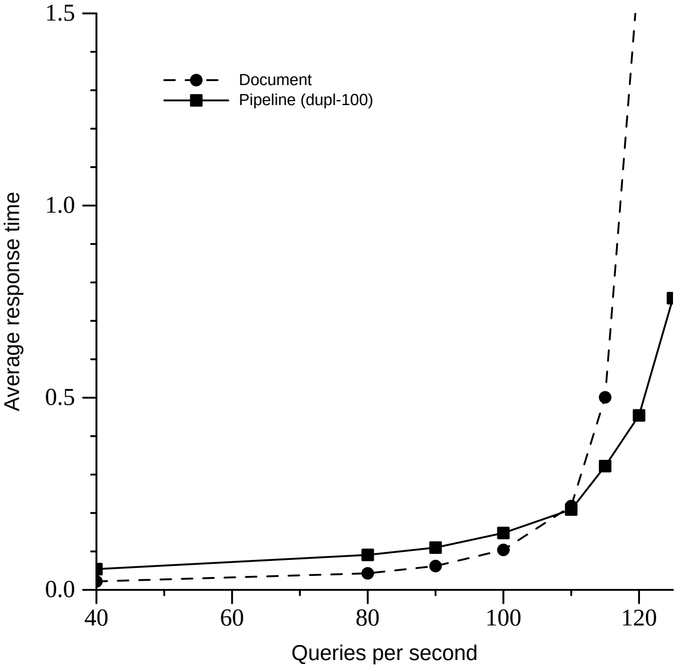


Figure 8.2: Average query response time for document-distributed and pipelined systems on a four-node index of TB/02*. Other settings are as for Figure 8.1.*

processing parallelism of the document distribution architecture, compared to the inherent serialism of pipelining. As the number of queries per second increases, so too does the average response time. Queries start to arrive before the previous ones have been finished; the system has to share resources between them, and so each one takes longer to process. The increase in average response time is similar for both architectures until quite heavy workloads are reached. At 110 queries per second, the document-distributed system is still processing each query in around two-thirds of the time of the pipelined system. However, by 120 queries per second, the document-distributed system is clearly at the very limits of its capacity, and the average query response time has shot up sharply; here, it is the pipelined system that is taking two-thirds of the time to answer each query. Pipelining's lower average response times at high system loads when compared to document distribution is a direct result of its higher query throughput capacity. Nevertheless, for most workloads, document distribution offers faster response times.

Another question of interest is what effect scaling the system has upon average response times at low to moderate workloads. Figure 8.2 is enlightening in this regard. Here, four nodes are being used to run an index of TB/02; that is, one step has been taken backwards along the diagonal of system scale.

Behaviour similar to Figure 8.1 is observed; document distribution has faster response times at low loads, with the curves crossing as document distribution hits its system capacity. There is, however, an instructive difference between the two graphs at low loads. Document distribution, with its parallelised processing, displays similar average response times for the two system sizes. In contrast, pipelining's inherently serialised processing causes its average response time to increase with system scale, being for the 8-node system roughly double what it is for the 4-node one. This is a major limitation of pipelining: average response time under light load increases roughly in proportion with system scale, whereas under document distribution it is more or less constant. In other words, increasing cluster size under pipelining will increase maximal throughput, but it cannot help resolve potential issues with response time under low to moderate loads.

8.4 Summary

This chapter has examined the scalability of the pipelined system when enhanced with partial replication and intelligent query routing. An unreplicated pipelined system with arbitrary term allocation suffers increasingly poor load balancing as the degree of partitioning increases, as was observed in Chapter 6. With partial replication and intelligent routing, however, the pipelined architecture is able to overcome this problem and record a slight increase in throughput as the system scales in size. In addition, if the memory available for disk caching is reduced, pipelining's superiority in throughput over document distribution becomes more marked, not only meaning that the deployer can save on memory costs, but also suggesting that pipelining will respond better to per-node index sizes larger than those used in this thesis. On the other hand, document distribution provides faster average response times than pipelining at low to moderate load levels, and maintains these fast response times as the system scales, whereas pipelining's low-load response times increase with system scale. Chapter 9, which concludes the thesis, will provide some guidance to the implementor of a real-world search engine on how to weigh the relative advantages of the two architectures.

Chapter 9

Conclusions and future directions

This chapter summarises the findings of the thesis, provides guidance for the system implementor on which distributed architecture to adopt, and presents some possible directions for future research in the area.

9.1 Conclusions

The focus of this thesis has been upon evaluating the throughput potential of the pipelined system. At the outset of the research described here, pipelining was a promising but untested new architecture for distributed full-text query evaluation, one that offered the prospect of combining the superior on-disk data contiguity of term partitioning with the shared processing model of document distribution. Contemporary popular web-scale search engines have immense processing loads placed upon them, and even a small improvement in efficiency can mean significant savings in hardware and running costs.

The experiments described in Chapter 6 indicated, however, that the pipelined architecture faced a serious problem with load balancing, one that overwhelmed the benefits to be derived from its superior disk access characteristics. In contrast, document distribution demonstrated excellent natural load balancing. While pipelining was a significant improvement upon the traditional term-distributed architecture, its load balancing problems clearly had to be resolved before it could be a serious challenger to document distribution.

A number of approaches to load-balancing a pipelined system were described in Chapter 7. Since the root of the problem is the skewed workload distribution amongst query terms, the obvious solution is to explicitly allocate terms to nodes so as to balance out the workload skew. Unfortunately, balancing workload through term allocation is not as straightforward as it might appear. The term frequency distribution in the query stream is not the same as in the document collection, and so a sample of the query stream must be used to arrive at reasonable estimates of future term workloads. The composition of the query stream varies from one sample to the next, however, so even with sampling-based term allocation strategies, long-term imbalances remain.

A more successful load-balancing strategy than explicit term allocation is partial term replication and intelligent query routing. Replication and routing are able to achieve perfect simulated long-term balance, and demonstrate good load balancing in experimental runs on an implemented system, too. In addition, term replication allows the system to route around short-term bottlenecks that occur due to the natural variability in term occurrences in the query stream. The results given in Chapter 7 indicate that a partially-replicated, intelligently-routed pipelined system is able to match and even marginally outperform document distribution, for the experimental configuration employed. The additional experiments described in Chapter 8 not only demonstrate the scalability of the partially-replicated pipelined system, but also indicate that it significantly outperforms document distribution in a memory-limited environment, which in turn suggests that it would likewise outperform were the per-node index size to be larger than that used in this thesis's experimental configuration. At the same time, it must be recognised that document distribution offers faster average response times at low to medium loads, and holds these response times more or less constant as the system scales, in contrast to the proportionate increase in response time that occur in the pipelined system.

The reliability of these findings rests upon the thoroughness of the experimental methodology employed, which was described in Chapter 4. A poorly-designed query set, for instance, could underemphasise the skewed nature of the term workload distribution, and therefore obscure the load balancing problems that this causes the pipelined system. Similarly, only by building scalability into the experimental design does the excellent natural scalability of the document-distributed architecture become apparent.

In light of the findings of this thesis, and given the assumption of document-ordered inverted lists and term-at-a-time processing, which is the distributed architecture of choice for the system implementor? The answer depends on the performance requirements and available resources of a particular deployment. If the slower average query response time is acceptable, then money can be saved by implementing a partially-replicated pipelined system on a cluster of machines with reduced memory, and running the cluster at or near its throughput capacity. If fast per-query response time is essential, however, the clear choice is to implement a document-distributed system, with generous amounts of memory on each node to maximise index caching and thus minimise the impact of the architecture's larger per-node vocabulary and fragmentation of inverted list information. In addition, the cluster should have enough nodes to ensure that the system is running well below full utilisation. A hybrid architecture would also be possible, in which the collection was document-partitioned to reduce average response time, and then each of these document-wise partitions was sub-partitioned by terms and deployed upon a pipelined cluster to squeeze the maximum performance out of the hardware budget.

9.2 Future directions

There are a number of directions that future research in this area could take. As noted in Section 7.3, the query routing algorithms examined for the partially replicated pipelined system are somewhat crude. An improved algorithm would build upon a more precise measurement of current workload at each node, and also use information about queries currently in process in the system which are yet to reach the node. Such an algorithm would improve short-term load balancing, thus squeezing the last few percentage points of utilisation out of the system. It would also be desirable, when the routing choice is between two fairly evenly loaded nodes, to choose to send the query to the node which had most recently handled the term, in order to improve cache effectiveness. Additionally, a more sophisticated analysis of query term co-occurrence in the historical query stream would enable fine tuning of the term assignment and replication strategies, although whether the goal should be to maximise the extent to which co-occurring query terms are held on the one node, to reduce the number of query bundle hops, or minimise co-occurrence, to reduce load imbalance, remains to be clarified.

The relatively poor query response time of the pipelined system under light to moderate workloads has been noted as a significant limitation of the architecture. Pipelining's response time could be improved by preloading all inverted lists for a query as soon as the query reaches the receptionist, rather than loading inverted lists only when the query bundle arrives at a node. Preloading inverted lists would enable disk parallelism. A degree of processing parallelism could be achieved by forwarding accumulators from one machine to the next in blocks as they are created, rather than waiting for processing at the originating node to be completed. Nevertheless, these methods, while decreasing query response time, cannot ultimately solve its non-scalability in the pipelined system, since the time to load and process the longest inverted list sets a lower bound on overall processing time.

The experiments in this thesis have all worked with document-ordered lists. Processing ranked queries with document-ordered lists requires that the full inverted list be loaded from disk. With web-scale collections, these inverted lists become quite large for frequent terms, and so the roughly fixed seek time is amortised across a long read time. The primary advantage of term partitioning, and hence of pipelining, is in the reduction in the number of seek operations; the long reads of loading full inverted lists dilute this advantage. There are alternative inverted index organisations, such as inverted lists containing and ordered by pre-computed impacts [Anh and Moffat, 2006a], in which only the front portion of each inverted list, which contains high-weight documents, needs to be read. The shorter reads of such environments would accentuate pipelining's advantage in reducing the number of seek operations.

There are also some interesting possibilities for increasing the throughput of document distribution. As described in this thesis, each node in a document-distributed system operates autonomously, with the possible exception that global rather than local term statistics are used. However, it should be possible to make use of the results of running a query on one or a subset of the nodes of a document-distributed cluster to inform the query evaluation carried out on the remaining nodes. For instance, the document scores derived on one node could be used to set a more strict accumulator threshold for the remaining nodes. If such an evaluation strategy required other nodes to delay processing until one or a subset of nodes had completed either a partial or a full evaluation of the query, then query response time would suffer, which might not be desirable or tolerable. However, random variability in processing scheduling due to disk location, caching, and network latency mean that it will naturally happen that some nodes will start processing a particular query before others, and it may be possible to opportunistically use the natural staggering in processing to inform some nodes of the partial results achieved on others.

9.3 Summary

The pipelined architecture is an interesting and novel approach to the efficient implementation of a distributed information retrieval engine. This thesis has undertaken a meticulous examination of this new architecture, and has sought to advance not only our understanding of its strengths and its limitations, but also the methodology by which experiments on the efficiency of distributed architectures are performed. The final assessment of the pipelined architecture is mixed, but there are clearly domains in which it has significant potential. The research presented in this thesis lays a firm foundation from which this potential might be realised.

Bibliography

V. N. Anh and A. Moffat. Impact transformation: Effective and efficient web retrieval. In K. Järvelin, M. Beaulieu, R. Baeza-Yates, and S. H. Myaeng, editors, *Proc. 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 3–10, Tampere, Finland, Aug. 2002.

V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Kluwer International Journal of Information Retrieval*, 8(1):151–166, Jan. 2005.

V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In S. Dumais, E. Efthimiadis, D. Hawking, and K. Järvelin, editors, *Proc. 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Seattle, WA, Aug. 2006a.

V. N. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE Transactions on Knowledge and Data Engineering*, 18(6): 857–861, June 2006b.

V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In W. B. Croft, D. J. Harper, D. H. Kraft, and J. Zobel, editors, *Proc. 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 35–42, New Orleans, LA, Sept. 2001.

C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In G. Navarro, editor, *Proc. 8th International Symposium on String Processing and Information Retrieval*, pages 10–20, Laguna de San Rafael, Chile, Nov. 2001.

R. Baeza-Yates. Web usage mining in search engines. In A. Scime, editor, *Web Mining: Applications and Techniques*. Idea Group Publishing, 2005.

R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, New York, New York, 1999.

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

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

S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In P. Enslow and A. Ellis, editors, *Proc. 7th International Conference on the World Wide Web*, pages 107–117, Brisbane, Australia, 1998. Elsevier Science Publishers B. V.

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. 26th European Conference on IR Research*, volume 2997 of *Lecture Notes in Computer Science*, pages 394–408, Sunderland, UK, Apr. 2004. Springer.

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. 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 110–118, Zurich, Switzerland, Aug. 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, Jan. 2000.

J. Callan. Distributed information retrieval. In *Advances in Information Retrieval: Recent Research from the Center for Intelligent Information Retrieval*, pages 127–150. Kluwer Academic Publishers, 2000.

J. Callan, F. Crestani, and M. Sanderson. SIGIR 2003 workshop on distributed information retrieval. *ACM SIGIR Forum*, 37(2), 2003.

D. Cutting and J. Pedersen. Optimization for dynamic inverted index maintenance. In *Proc. 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 405–411, New York, NY, USA, 1990. ACM Press.

D. D'Souza, M. Fuller, J. Thom, P. Vines, J. Zobel, O. de Kretser, R. Wilkinson, and M. Wu. Melbourne TREC-9 experiments. In E. M. Voorhees and D. K.

Harman, editors, *Proc. TREC-9*, pages 437–451, Nov. 2000. NIST Special Publication 500-249.

E. Fox, D. Harman, and R. Baeza-Yates. Inverted files. In W. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, chapter 3. Prentice-Hall, 1992.

A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 902–903, New York, NY, USA, 2005. ACM Press.

D. Harman. Overview of the first TREC conference. In *Proc. 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 36–47, New York, NY, USA, 1993. ACM Press.

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

B. P. Jansen, A. Spink, J. Bateman, and T. Saracevic. Real life information retrieval: A study of user queries on the web. *ACM SIGIR Forum*, 32(1): 5–17, Spring 1998.

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.

T. Joachims, L. Granka, B. Pan, H. Hembrooke, and G. Gay. Accurately interpreting clickthrough data as implicit feedback. In G. Marchionini, A. Moffat, J. Tait, R. Baeza-Yates, and N. Ziviani, editors, *Proc. 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Salvador, Brazil, Aug. 2005.

M. Kaszkiel and J. Zobel. Term-ordered query evaluation versus document-ordered query evaluation for large document databases. In W. B. Croft, A. Moffat, C. J. van Rijsbergen, R. Wilkinson, and J. Zobel, editors, *Proc. 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 343–344, Melbourne, Australia, Aug. 1998.

J. Kekäläinen and K. Järvelin. Using graded relevance assessments in IR evaluation. *Jour. of the American Society for Information Science and Technology*, 53(13):1120–1129, Sept. 2002.

M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30:81–93, June 1938.

N. Lester. *Efficient index maintenance for text databases.* PhD thesis, RMIT University, Australia, 2006.

N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In A. H. Ngu, M. Kitsuregawa, E. J. Neuhold, J.-Y. Chung, and Q. Z. Sheng, editors, *Proc. 6th Int. Conf. on Web Informations Systems*, pages 470–477, New York, Nov. 2005. LNCS 3806, Springer.

M. Levene and A. Poulovassilis. Web dynamics. *Software Focus*, 2(2):60–67, 2001.

Z. Lu and K. S. McKinley. Partial collection replication versus caching for information retrieval systems. In E. Yannakoudis, N. Belkin, M. Leong, and P. Ingwersen, editors, *Proc. 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 248–255, Athens, Greece, Aug. 2000.

Z. Lu and K. S. McKinley. Partial replica selection based on relevance for information retrieval. In M. Hearst, F. Gey, and R. Tong, editors, *Proc. 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 97–104, San Francisco, CA, Aug. 1999.

A. MacFarlane, J. A. McCann, and S. E. Robertson. Parallel search using partitioned inverted files. In P. de la Fuente, editor, *Proc. 7th International Symposium on String Processing and Information Retrieval*, pages 209–220, A Coruña, Spain, Sept. 2000.

A. Moffat and J. Zobel. What does it mean to "measure performance"? In X. Zhou, S. Su, M. P. Papazoglou, M. E. Owlowska, and K. Jeffrey, editors, *Proc. Fifth International Conference on Web Information Systems*, pages 1–12, Brisbane, Australia, Nov. 2004. LNCS 3306, Springer.

A. Moffat and J. Zobel. Rank-biased precision for measurement of retrieval effectiveness. 2006. Submitted for review.

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

A. Moffat, J. Zobel, and R. Sacks-Davis. Memory efficient ranking. *Inf. Process. Manage.*, 30(6):733–744, 1994.

A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In S. Dumais, E. Efthimiadis, D. Hawking, and K. Järvelin, editors, *Proc. 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 348–355, Seattle, WA, Aug. 2006a.

A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Kluwer International Journal of Information Retrieval*, Oct. 2006b. Published online.

M. J. Moroney. *Facts from Figures.* Penguin Books, New York, 1951.

S. Orlando, R. Perego, and F. Silvestri. Design of a parallel and distributed web search engine. In *Proc. 2001 Parallel Computing Conference*, pages 197–204. Imperial College Press, Sept. 2001.

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

E. Rasmusen. Introduction: parallel processing and information retrieval. *Information Processing & Management*, 27(4):255–263, 1991.

B. Ribeiro-Neto, E. S. de Moura, M. S. Neubert, and N. Ziviani. Efficient distributed algorithms to build inverted files. In M. Hearst, F. Gey, and R. Tong, editors, *Proc. 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 105–112, San Francisco, CA, Aug. 1999.

B. A. Ribeiro-Neto and R. R. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proc. 3rd ACM Conference on Digital Libraries*, pages 182–190, Pittsburgh, PA, June 1998. ACM Press, New York.

S. E. Robertson, S. Walker, M. M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In D. K. Harman, editor, *Proc. TREC-3*, pages 109–126, Nov. 1994. NIST Special Publication 500-225.

G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *ACM Transactions on Information Systems*, 18(11):613–620, 1975.

M. Sanderson and J. Zobel. Information retrieval system evaluation: effort, sensitivity, and reliability. In G. Marchionini, A. Moffat, J. Tait, R. Baeza-Yates, and N. Ziviani, editors, *Proc. 28th Annual International ACM SIGIR*

Conference on Research and Development in Information Retrieval, pages 162–169, Salvador, Brazil, Aug. 2005.

F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In K. Järvelin, M. Beaulieu, R. Baeza-Yates, and S. H. Myaeng, editors, *Proc. 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 222–229, Tampere, Finland, Aug. 2002.

L. Si and J. Callan. Modeling search engine effectiveness for federated search. In G. Marchionini, A. Moffat, J. Tait, R. Baeza-Yates, and N. Ziviani, editors, *Proc. 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 83–90, Salvador, Brazil, Aug. 2005.

C. Silverstein, H. Marais, M. Henzinger, and M. Moricz. Analysis of a very large web search engine query log. *ACM SIGIR Forum*, 33(1):6–12, 1999.

I. Soboroff. Does WT10g look like the web? In K. Järvelin, M. Beaulieu, R. Baeza-Yates, and S. H. Myaeng, editors, *Proc. 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 423–424, Tampere, Finland, Aug. 2002.

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

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

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

H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Information Processing & Management*, 31(6):831–850, 1995.

E. Voorhees and D. Harman, editors. *TREC: Experiment and Evaluation in Information Retrieval*. The MIT Press, 2005.

E. M. Voorhees. Evaluation by highly relevant documents. In W. B. Croft, D. J. Harper, D. H. Kraft, and J. Zobel, editors, *Proc. 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 74–82, New Orleans, LA, Sept. 2001.

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, Dec. 2005.

H. Williams and J. Zobel. Searchable words on the web. *International Journal of Digital Libraries*, 5(2):99–105, 2005.

I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, San Francisco, second edition, 1999.

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 Conference on Digital Libraries*, pages 422–431, Rome, Sept. 2002. Springer. LNCS volume 2458.

G.-R. Xue, H.-J. Zeng, Z. Chen, Y. Yu, W.-Y. Ma, W. Xi, and W. Fan. Optimizing web search using web click-through data. In *Proc. 13th Conf. on Information and Knowledge Management*, pages 118–126, New York, NY, USA, 2004. ACM Press.

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

J. Zobel. How reliable are the results of large-scale information retrieval experiments? In W. B. Croft, A. Moffat, C. J. van Rijsbergen, R. Wilkinson, and J. Zobel, editors, *Proc. 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 307–314, Melbourne, Australia, Aug. 1998.

J. Zobel and A. Moffat. Exploring the similarity space. *ACM SIGIR Forum*, 32 (1):18–34, 1998.

J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4):453–490, 1998.