DISTRIBUTED INVERTED FILES AND PERFORMANCE: A STUDY OF PARALLELISM AND DATA DISTRIBUTION METHODS IN IR

ANDREW MACFARLANE

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

at

City University, London

Department of Information Science

August 2000

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

ACKNOWLEDGEMENTS

To both Prof. S.E. Robertson and Dr J.A. McCann for their excellent supervision and advice over the course of this PhD. I am particularly grateful for Prof. Robertson's vast knowledge of the Information Retrieval arena and Dr McCann's deep knowledge of performance and parallelism.

To Dr David Hawking for many discussions and pointers in the area of parallelism for Information Retrieval and for both suggesting and making the arrangements for a visiting research studentship at the Australian National University in Canberra.

Thanks goes to Steve Walker who helped me understand much of the concepts behind the Okapi system, particularly methods used for passage retrieval and routing/filtering.

To my parents W.W. MacFarlane and B. MacFarlane and the rest of my family for their enthusiastic support during my studies.

To all the friends I made during my three month visit to the ANU and to all the staff in the Computer Science Department for making my stay in Canberra a memorable one.

To AcSys, CAP at the ANU and Imperial College, London for the use of their Alpha Farm, AP3000 and AP1000 parallel machines. I would also like to thank the staff of these institutions, in particular David Sitsky for his patient help and advice when it was needed.

To Microsoft Research, Cambridge for the use of their Pentium PII cluster "The Cambridge Cluster" and to all the staff at MSR for six wonderful months in the Cambridge Lab.

To both the British Academy and the DFEE for providing me with a grant that enabled me to complete my research.

To all my colleagues in room A420B at City University for their friendship.

ABSTRACT

The study investigates the performance of parallel information retrieval (IR) algorithms on different data distribution methods for Inverted files to identify which is the best for the requirements of specific IR tasks. We define a data distribution method as a way of distributing Inverted file data to local disks on a parallel machine. A data distribution method may be on-the-fly (with one copy of the index held), replication (all nodes have all of the index) or partitioning (data for index is split amongst nodes). Partitioning of inverted file data can be done in many ways but we consider only two: by term (*TermId*) and by document (*DocId*). *TermId* partitioning is a type of partitioning which distributes unique word data to a single partition, while *DocId* partitioning distributes unique document data to a single partition. We consider the issue of improving the performance of standard IR algorithms on these data distribution methods by looking at sequential job service not concurrent job service, e.g. we consider the issue of sequential query service not concurrent query service. This methodology rules out some distribution methods for some tasks studied. We consider the following main tasks of IR: indexing, search, passage retrieval, inverted file update and query optimisation for routing /filtering. We produce a synthetic performance model for each of these tasks for the purposes of comparison. We have two subsidiary aims; one was to demonstrate portability of our implemented data structures and algorithms on different parallel machines. Secondly, we also study the possibility of increased retrieval effectiveness by examining a larger section of the search space for both passage retrieval and routing/filtering. We consider the implications of concurrency in updates on Inverted files. Our theoretical and empirical results show that in most cases the *DocId* partitioning method is the best data distribution method apart from routing/filtering where replication was found to be superior.

Chapter 1

Introduction

1.1 THESIS BACKGROUND

The growth of the Internet and the web has increased the interest in information retrieval (IR) particularly on searching the web. While doubts have been cast on the availability of search over the whole web (Lawrence and Giles, 1999), many web search engines still index around 1.25 billion pages. There is a need to service queries over these very large collections providing fast access for individual users as well as fast processing for large volumes of users. Many information providers such as Dialog, Reuters or FT profile allow web access to their collections and can expect an increasing interest in their services as the web grows.

Our motivation for the research to be presented in this thesis is to tackle some of the problems of handling large data sets over a number of different IR tasks. We define a task as being a specific aspect of an IR system that has its own functionality. Some tasks may use others (a super-set), but may also be discrete. The tasks to be studied are;

- Indexing.
- Probabilistic search.
- Passage retrieval.
- Routing/filtering.
- Index update.

Each of these tasks is defined individually below (see section 1.4). The effects of large data sets on these tasks differ as some are more computationally intensive than others, e.g. indexing 500 million documents will take longer than serving a query over it. Some tasks are poorly understood such as index update, particularly within the context of concurrent update and query transaction processing.

Much of the previous work on large data sets uses parallel computing, an approach that we advocate strongly (see chapter 2). Some of the most successful research in this area concentrates on searching the data using inverted files with some interest in indexing. There has been very little discussion on using parallel computing for the other tasks under consideration, particularly within the context of data distribution methods. We define data distribution as the allocation of inverted file data to nodes in a parallel computer. A survey of data distribution methods for the IR tasks is needed within a holistic framework: a basic question to ask is which is the best data distribution as a whole. We do this by examining data distribution on each task in turn and stating which is the best method demonstrated by theory and experience, building up the whole picture bit by bit.

1.2 AIM AND INTENTIONS OF THE THESIS

Our intentions in the thesis are as follows. We focus on textual IR and do not study any other type of media such as speech, images or multimedia. We look at data distribution methods as the impact of distribution methods on performance in parallel IR systems is a significant factor. Our primary aim is to examine the issue of data distribution and performance as a whole on more than one IR task (introduced above) by examining theoretical and empirical results on inverted files using parallel computers. We consider the issue of speeding or scaling up an algorithm in a batch job mode, such that only one discrete body of work is completed within a given time period. The primary questions to be addressed in this thesis are as follows. How can we apply parallel methods to IR systems in order to improve the performance and efficiency of such systems? How does a given data distribution method affect the performance of different IR tasks? What is the best data distribution method for a given task and for all tasks?

We also have some secondary aims. One is to demonstrate portability in our algorithms: much of the previous work done in parallel computing for IR was machine specific. We investigate the possibility of enhanced retrieval effectiveness in passage retrieval and routing/filtering by examining more of the search space in those tasks. The effect of query size on probabilistic search is investigated. With respect to index update we examine the practicalities of using partitioned inverted files and discuss the viability of such.

Because of time and resources we do not address some very important issues. We do not examine the issue of concurrent query or transaction service in the probabilistic search or passage retrieval tasks, and only address the issue theoretically in the index update tasks (concurrent service does not apply to the routing/filtering or indexing tasks). We do not examine the use of query processing optimisation techniques for speeding up query service (Hawking, 1998), which has the side effect of reducing retrieval effectiveness depending on the level of optimisation used. We do not directly address these issues empirically in the thesis but we recognise their importance and discuss them where necessary.

1.3 DEFINITION OF DISTRIBUTION METHODS

We look at four distribution methods; on-the-fly distribution, replication and two types of partitioning. Because of the criteria used some data distribution methods are invalid for some tasks (this issue is tackled when the tasks are defined below). In order to explain the differences between these distributions we use an abstract task, which is defined as follows. We have some inverted file data D, together with some work W to be done on D (W can be any information retrieval task e.g. index a document or do a search). When W is applied to D, we get a result R. Any or all of the three variables D, W and R may be subdivided or paritioned in some way; W', D' and R' will denote some such partition or subset, which may nevertheless be the whole of the orginal variable in some cases. We define some algorithmic steps on these variables which are common to all distribution methods;

- A central node sends W' plus any data needed to a number of i identical processor nodes.
- The processor nodes apply W' to D' to produce results R'.
- The results R' are sent back to the central node which prepares the final result R using all R' results.

It may help the reader to think of an inverted file (or more formally D) as a matrix with the rows made up of term references and the columns made up of references to documents, see fig 1-1.

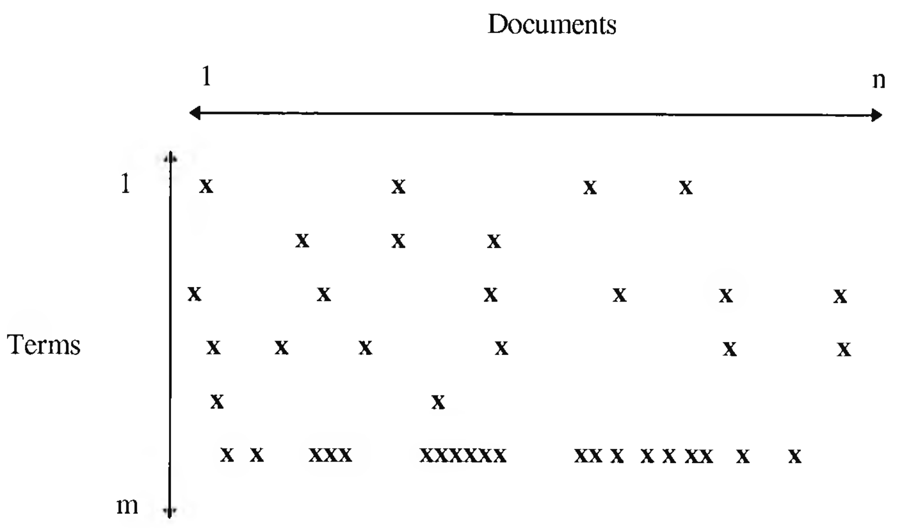


Fig 1-1: Example matrix

In the example we have m terms in the collection which invert n documents from the indexed text. Relations between terms and documents are signified by an **x**. The matrix will be very sparse, and some relations between documents and terms or vice versa will be richer than others. We will use this matrix representation to show how data is distributed in the methods to be examined in this thesis.

1.3.1 *On-the-fly distribution*

We define on-the-fly distribution as the distribution of part of the data as it is required by a given task. It is a dynamic data distribution method, all the others are static ones. The method is very flexible in that we can distribute the whole matrix D or any D' which could be the whole or part of either a row or column. The inverted file data is held in one location. Consider the following example;

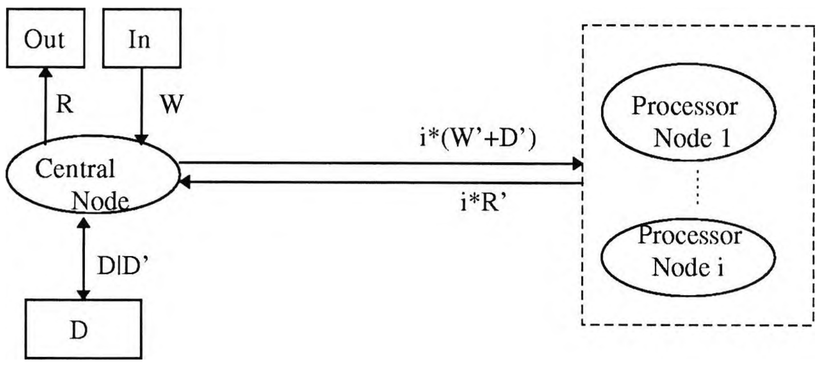


Fig 1-2: Example of abstract task on On-the-fly distribution

Note that the ellipse is a node, a box is a disk, and the dashed line box is the universe of our processor nodes. The arrows signifies the direction of data exchange between nodes and or disks. Our abstract task uses the following algorithmic steps. The work W is input and the data associated with W (D') is lifted from the inverted file and is packed up with a subset of W (W'). Each of the i processor nodes each gets its own W' and D', and processes the data producing R'. All results from i processor nodes (R') are sent back to the central node which prepares the final result R. The significant disadvantage with this method is that a large amount of data must be distributed before computation can be done. The communication may swamp any useful work, and this makes the distribution method impractical for many information retrieval tasks.

1.3.2 *Inverted file replication*

Replication is the duplication of inverted file data on local disks of a parallel computer. Each node in the system has access to all the data locally, therefore the need to transmit large amounts of data is greatly reduced. The advantage of replication as against On-the-fly distribution is that the high communication cost is much reduced without loss of flexibility. The disadvantage is that space costs are considerably higher than any of the other distribution methods discussed in this thesis. Consider the following example;

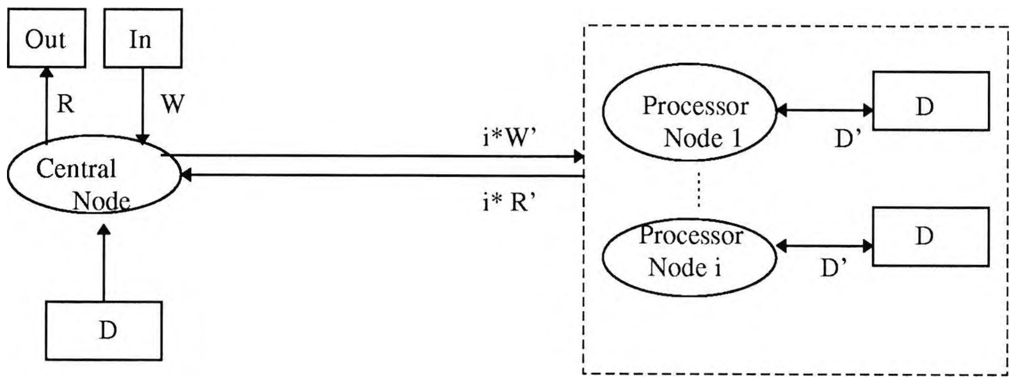


Fig 1-3: Example of abstract task on inverted file replication.

The key issue here is that all nodes have access to matrix D locally. In our abstract task we send each processor node W' together with some scheme for partitioning the matrix as required by the given computation or task. We then produce R' for each processor node and return the result back to the central node to compute the final R as would be done with on-the-fly distribution. A further advantage in having the whole matrix D available to the node is that load can be re-balanced by exchanging subsets of W' between the processor nodes, without having to communicate any aspect of D.

1.3.3 *Inverted file partitioning*

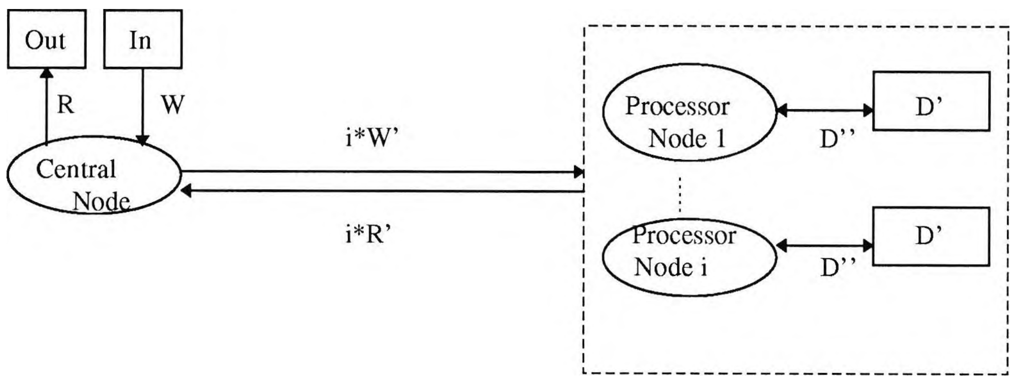


Fig 1-4: Example of abstract task on inverted file partitioning.

Partitioning is the fragmentation of inverted file data over local disks in a parallel computer (see fig 1-4). In the example given in figure 1-4 each node has access to its own subset of D, D' and which can only be accessed by that node. In terms of the abstract task, each node manipulates a subset of D', D'' in order to service work W or W'. The node services W or W' depending on the task and partitioning type. The advantage of partitioning is that the space costs are lower than replication, but it is a static distribution method and is therefore not as flexible as either replication or On-the-fly distribution of inverted files. The process of distribution is also much more complex than inverted file replication.

There are two main inverted file partitioning methods (Jeong and Omiecinski, 1995): by term identifier (*TermId*) and by document identifier (*DocId*). These partitioning methods are orthogonal to each other. With *DocId* partitioning the terms for a single document are placed on one disk, therefore postings for the same term may be held on multiple disks (see fig 1-5).

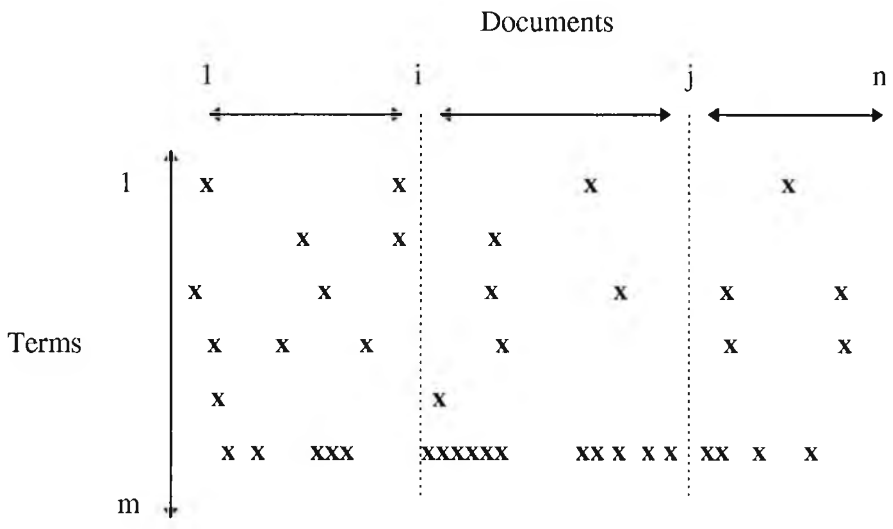


Fig 1-5: Example *DocId* matrix partitioning

We assume for arguments sake that we have three partitions in our example. In the example documents 1 to i-1 are given to node 1, documents i to j-1 are given to node 2 and documents j to n are given to node 3. The demarcation of the partitions is signified by the dotted line. With respect to our abstract task, we need to distribute W (if W is a query) to all partitions as all may have data for any or all of the terms in W. With *TermId* partitioning however, all postings for a given term are on one disk, therefore postings for the same document may be on multiple disks (see fig 1-6). In the example terms 1 to i-1 are given to node 1, terms i to j-1 are given to node 2 and terms j to n are given to node three. With respect to our abstract task and our specific example given above on *DocId*, nodes get there own unique subset of work W' as each nodes has its set of unique terms.

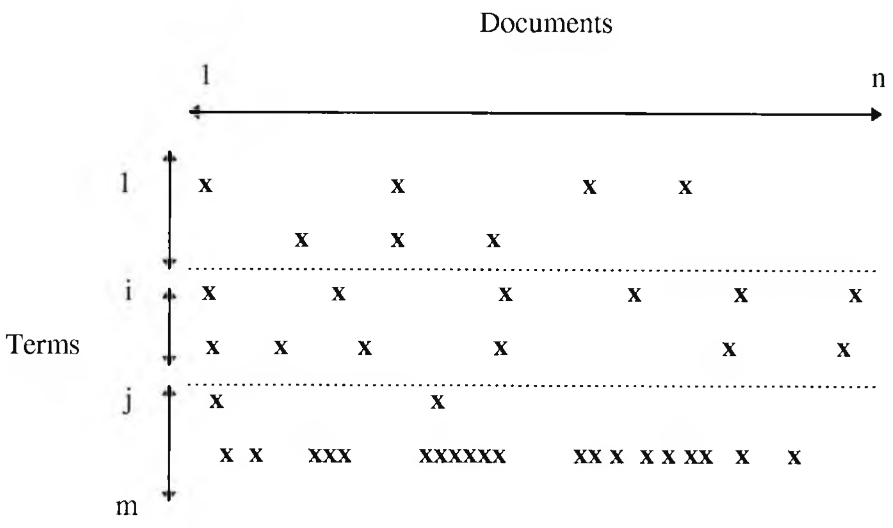


Fig 1-6: Example *TermId* matrix partitioning

1.4 INFORMATION RETRIEVAL TASKS STUDIED

We have defined a task as being a specific aspect of an IR system. We do not attempt to examine every task in IR as the field is large. Thesaurus construction, clustering and hypertext creation tasks are among the notable exceptions which we do not investigate. We largely concentrate on what we regard as the main tasks in IR such as indexing and search: we define a main task as one with which an IR system using inverted files would be unable to function if such did not exist. The other (non-main) tasks studied can be built using the core tasks and extended as required. For example index update can be built from search and index functionality, while routing/filtering and passage retrieval can be built from search functionality. We chose the non-main tasks based on the amount of research done on them in the past using parallel techniques (see chapter 2). Time restrictions prevented us from investigating many other tasks of IR. Each of the tasks to be studied in this thesis is defined below.

1.4.1 *The indexing task*

The indexing task is the process of taking raw text and building an index over that text using some criteria such as removal of stop words and stemming, etc. The index we built is the inverted file method (described in chapter 3). The process of indexing is one of the most computationally intensive aspects of IR requiring vast CPU, memory and disk resources (but is done only once and incremented thereafter). It is therefore a prime candidate for the application of parallelism. We consider partitioning only for this task. On-the-fly distribution is irrelevant (our consideration of distribution is on inverted file data not raw text). Replicated indexes can be produced by indexing on one processor and copying the data to the nodes.

1.4.2 *The probabilistic search task*

The probabilistic search task is the process of servicing queries on inverted files to produce a ranked list of documents using a term weighting scheme such as that derived by Robertson/ Sparck Jones (1976). Query processing in this context is very fast, which is why inverted files in some form have become the dominant storage technique in IR. We will show that it is still possible to increase the speed of query processing further using parallelism, and by doing so increase the system throughput. With respect to distribution methods, replication is not a suitable method for probabilistic search (with the possible exception of concurrent query service) while on-the-fly would restrict efficiency due to the extra communication overhead. It is likely that this extra overhead would outweigh any gain made by parallelism. Our discussion on the probabilistic search task is restricted to partitioning methods only.

1.4.3 *The index update task*

The index update task consists of a number of different aspects. We consider the issue of transaction processing where a transaction is either a document to be inserted or a probabilistic search request. We define index update as data to be periodically added to the index when a buffer with document insertions has exceeded some memory limit. This requires some form of index reorganisation which must be done concurrently with transaction processing to prevent delays: we assume that transaction processing cannot be suspended and specify a requirement that queries should be serviced as soon as possible. It should be noted that we consider insertions only, not deletions or modifications. To do otherwise would complicate the process further and in any case most text collections are archival in nature (the web being a notable exception). Index maintenance is a computationally intensive activity and we study the use of parallelism to speed the process up and to make it viable in a transaction processing context. We study partitioning methods only for the same reasons as given for the probabilistic search task.

1.4.4 *The passage retrieval task*

The passage retrieval task as applied in this thesis is from Okapi experiments conducted within the TREC conference framework, in particular Okapi at TREC-3 (Robertson et al, 1995). We classify passage retrieval as being the retrieval of part of a document that is most likely to be of interest to a user. It should be noted that this includes the whole document itself. Retrieval is done on the basis of text atoms, which can be paragraphs (Robertson et al, 1995), blocks, sentences, words (Kaszkiel and Zobel, 1997) or even characters. A passage is usually defined as a contiguous sequence of atoms, and new passages may be generated iteratively from old ones by adding or removing blocks of atoms in increments. Once the text atom and its incremental level are defined we can either define static or arbitrary lengths for passages based on the atom and the increment size. A fixed length passage starts from a given atom position, and computes the passage weight using n atoms (Callan, 1994). An arbitrary length passage mechanism relaxes this constraint and allows computation of passage weight for a passage of arbitrary length. This is rather expensive computationally (Robertson et al, 1995; Kaszkiel and Zobel, 1997) requiring $O(n^3)$ steps. We can compromise between the two methods and have maximum length passages, reducing the time complexity to $O(n^2)$: this is the method we use. A further refinement is the choice of increment. We may disallow overlapping passages to reduce the computation even further, but we do compute on overlapping passages. We determine both passages and atoms at indexing time. Passage processing is very computationally intensive. For example it is reported by Kaszkiel and Zobel (1997) that a run on 2 GBs of text with 50 queries using their passage retrieval techniques may take about one week on a SPARC 20. There is obviously much room for improvement, and parallel computing has the potential to reduce this time cost. We study partitioning methods only for the same reasons as given for the probabilistic search task.

1.4.5 *The routing/filtering task*

The idea behind information filtering is to disseminate incoming documents to users who require them. Users have long term information needs which may be satisfied by newly published documents. A method for information filtering is to take the documents that have been marked relevant by the user in the past and apply a relevance feedback mechanism to obtain a set of terms that can be applied to the new documents. We use TREC definitions of routing and filtering (Harman, 1996). In routing we provide the user with the top *n* documents, while in filtering we make a binary decision on n documents as to which will be presented to the user. There are a number of filtering techniques: batch filtering that takes n documents as a batch and adaptive filtering where documents are considered one at a time, with the possibility of feedback after each one. We consider batch filtering only. In Robertson et al (1995) it was stated that an alternative to some term ranking methods described would be to "evaluate every possible combination of terms on a training set and use some performance evaluation measure to determine which combination is best". Such a method is very computationally intensive, and certainly not practicable even with parallel machinery. For example, it was reported in Okapi at TREC-4 (Robertson et al, 1996), that term selection on 50 terms for 25 topics took between 4 and 15 hours on a Sun SS10 (for 50 topics with 200 terms the elapsed time can be up to a week) depending on the scoring/selection method used. Given this computational complexity, there is plenty of scope for further improvement. A further incentive was the success Okapi has had at TREC using these query optimisation techniques. Is it possible to examine more of the search space using parallelism and thereby increase retrieval effectiveness? We examine all the data distribution techniques theoretically, but have implemented only on-the-fly and replication methods for reasons that will become apparent.

1.5 MAIN FINDINGS OF THE THESIS

We outline the main findings of this thesis based on the aims and objectives declared above and the information retrieval tasks we consider;

- For most tasks, *DocId* partitioning is the best data distribution method for improving retrieval efficiency performance using parallelism. In particular, parallel computing using this partitioning method can make the use of inverted files for the index update task viable, depending on insertion rate.
- The only exception to the core finding was that replication was a better data distribution method for routing/filtering task.
- There is no evidence that searching more of a given space yields better retrieval effectiveness in the passage retrieval and routing/filtering tasks.

1.6 OUTLINE OF THE THESIS

The thesis is divided into eleven chapters and includes appendices and a glossary for reference. In the next chapter we give some background in the use of parallel computing for IR, identifying various areas of concern that have not yet been addressed. The methods, data and performance metrics used throughout the thesis are described in chapter 3. We describe various design and implementation details in chapter 4 by giving an extended description of the parallel IR system written for this thesis, namely PLIERS. In chapter 5 we use a synthetic model of performance in order to compare the performance of the various tasks theoretically and discuss the practicality of data distribution methods for those tasks. Relevant data distribution methods are identified for particular tasks that are then used for the purposes of the thesis. The rest of the thesis discusses empirical results for the most part. The general format is to discuss a data distribution method using some initial data set, and then to describe our TREC-8 experiments using the best data distribution method found either empirically or by theory. Index update is the exception to this as there is no relevant TREC track or task for that aspect of our thesis. Chapter 6 describes the performance of indexing on the partitioning methods in detail, while chapter 7 describes probabilistic search on those same methods. This is followed by a discussion on index update in chapter 8 that not only describes results on the two partitioning methods under consideration, but also gives a discussion on the problems of concurrency control mechanisms in a transaction service on inverted files. Chapter 9 and 10 discusses passage retrieval and routing/filtering results with more of an emphasis on retrieval effectiveness, particularly within the context of examining more of the search space. We give a summary and conclusion at the end of the thesis that tackles the issues of choosing an approach for a task and further research. A bibliography of referenced text is also provided.

Footnote: Some of the material presented in this thesis has been published earlier. This includes the review of parallel computing in IR (MacFarlane et al, 1997), PLIERS system description chapter (MacFarlane et al, 1999a), the probabilistic search chapter (MacFarlane et al, 2000b), and the theoretical part of the index update chapter (MacFarlane et al, 1996). All these publications were refereed. We also have had material published in unrefereed TREC publications (MacFarlane et al, 1999b; MacFarlane et al, 2000a) which makes up part of the index, probabilistic search, passage retrieval and routing/filtering chapters. All these papers were written by the author, with advice from his supervisors Prof. S.E. Robertson and Dr. J.A. McCann.

Chapter 2

Review Of Parallel Computing in Information Retrieval

2.1 INTRODUCTION

In this chapter we chart the progress of the use of parallel computing in information retrieval published in MacFarlane et al (1997) and updated here, following on from a review of the subject by Rasmussen (1992). We review important work in the past. We describe parallel architectures used for parallel IR systems. We analyse the different approaches to parallel IR using a classification due to Rasmussen (1992). Examples of parallel IR systems or methods are given in a case studies section. The motivation for the use of parallel computing in IR is an important strand in this chapter, in particular when and when not to use parallel systems. The retrieval models used in parallel IR systems are described. We give a summary at the end, and conclude by identifying areas not tackled in previous research to be addressed in the main body of this thesis.

2.2. PARALLEL ARCHITECTURES USED IN IR SYSTEMS

2.2.1 Parallel architecture classification

Flynn (Flynn, 1972) describes a taxonomy for classifying parallel architectures. A number of criticisms have been levelled at the taxonomy, e.g. there is no treatment of input/output and the instruction set used is ignored. In the context of IR, ignoring input/output is a particular problem (see section 2.2.3). In spite of these limitations the taxonomy has become the most popular method for describing parallel architectures and continues to be widely used in the field of parallel computing research including parallel IR. An alternative taxonomy is given by (Hockney and Jesshop, 1988). The Flynn taxonomy uses the concept of streams (Flynn, 1972) which are a sequence of items operated on by a CPU. These streams can either be *instructions* to the CPU or *data* to be manipulated by the instructions. We therefore have four broad classes of architecture:

(a) SISD - Single Instruction Single Data Stream;
(b) MISD - Multiple Instruction Single Data Stream;
(c) SIMD - Single Instruction Multiple Data Stream;
(d) MIMD - Multiple Instruction Multiple Data Stream.

The first of these, SISD is the normal sequential von Neumann architecture machine which has dominated computing since its inception. The MISD class is controversial: some argue that it is a null class and does not usefully describe any architecture (Deitel, 1990; Tanenbaum, 1990) while others assert that systolic arrays can be placed in this class (Hwang, 1993). We address the MISD class in our discussion on special parallel hardware below. We will ignore the SISD class for the rest of the chapter.

The SIMD class describes an architecture in which the same instructions operate on different data in parallel. It is therefore widely known as data parallel computing. Instructions are broadcast to n processors in the architecture which operate on the data held in that processor. Examples of this type of architecture are the ICL/CPP DAP (Bale et al, 1990) and Thinking Machines CM-2 (Hwang, 1993): the DAP is described in more detail below. The architecture had been dominant in the use of parallel computing in information retrieval until recently.

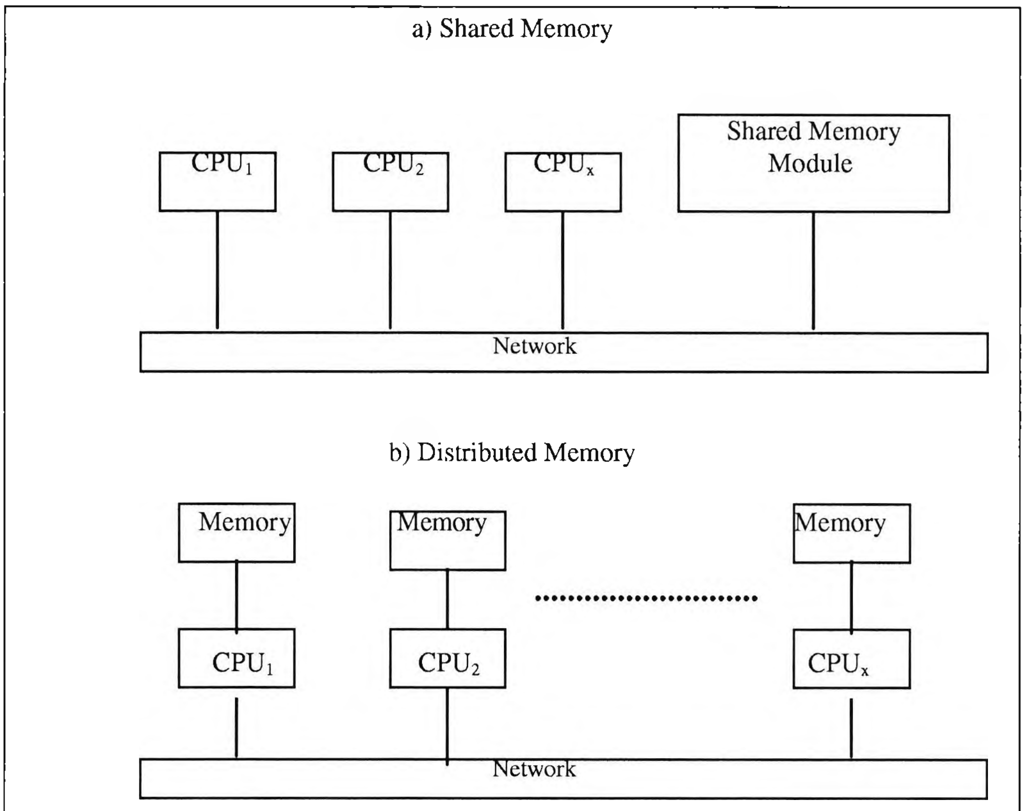


Fig 2-1. Types of memory organisation examples

The MIMD class describes an architecture in which processors independently execute different instructions on different data. The programs which run on this class of machine are therefore a great deal more complex than one could envisage on any of the other architectures.

There is a wide variety of this class of architecture including those where processors share the same memory and others in which processors have their own memory. These are known are shared memory and distributed memory architectures (see examples in fig 2-1). Each has its own subdivision which we will not attempt to describe here.

With the former, interprocessor communication is done through concurrency control mechanisms such as semaphores, while the latter uses message passing. There is also a hybrid architecture known as distributed shared memory (DSM) where programs see a single memory, but access is serviced by message passing. An example of a machine with the MIMD class architecture is the Fujitsu AP1000 which is described in section 2.2.2 below.

It should be noted that a further class of architecture exists which does not fit well into Flynn's classification. Special-Purpose Hardware has been built to accommodate IR systems (Hollaar, 1991) including associative memories, finite state machines and cellular arrays (Hollaar, 1992). Some of this work has been in building special purpose parallel architectures (Hurson et al, 1990) for text retrieval and we include it in the review for completeness

2.2.2 Parallel architectures used in IR

We now turn to specific machine architectures which have been used for parallel IR systems. We give an example of each type of architecture from section 2.2.1; the DAP, Fujitsu AP1000, and special parallel hardware. We also discuss the growing impact of networked workstation technology. More information on various architectures can be found in Rasmussen (Rasmussen, 1992).

I). *DAP (Distributed Array of Processors).* The CPP (formally ICL) DAP is a SIMD class architecture. The DAP organisation (Bale et al, 1990) is an array of 1-bit processing elements (PEs) arranged in a 32 by 32 matrix for the 500 series and 64 by 64 for the 600 series; 1024 and 4096 PE's in total respectively. The 600 series has four times the memory and processing power of the 500 series. Each processor is connected to its north, south, east and west neighbour processors (known as a NEWS grid) and to the row and column of the matrix by a bus system. Each processor has at least 32 Kbits of its own local memory. The ICL DAP needed a mainframe as a front end, but workstations can be used for current varieties. The architecture has a master control unit (MCU) which broadcasts instructions and data to the array to work on and also obtains the results from the array. The DAP has very fast I/O capabilities of up to 50 Mbytes per second to overcome the I/O bottleneck (the I/O problem in parallel computing for IR is discussed in section 2.2.3 below). The DAP is successfully used by the DapText system described by (Reddaway, 1991) and is included in the case studies section (2.6.1) below. Reuters use this system to provide Text Retrieval services to their customers. DapText has been implemented on both the 500 and 600 series of the DAP. Other work includes a British Library project for using the DAP in IR, described in (Pogue and Willett, 1987a and 1987b; Carroll et al, 1988; Pogue et al, 1988)

II). *Fujitsu AP1000.* The Fujitsu AP1000 is a MIMD distributed memory architecture with up to 1,024 SPARC processors or cells which are interconnected using a two dimensional torus network (this network looks rather like a flattened donut). Each cell can support from 16 Mbytes to 64 Mbytes per cell. Data can be moved in and out very quickly using a 50Mbyte per second broadcast network. To overcome the I/O bottleneck, the HiDIOS file system is useful with a load rate in excess of 50 Mbytes per second. The AP1000 has global reduction facilities which are useful for term weighting calculations. Work on IR using the AP1000 was pursued at the Australian National University through the PADRE system (Hawking, 1994 and 1995: Hawking and Thistlewaite, 1994 and 1996; Bailey and Hawking, 1996; Hawking et al, 1995), which has evolved through the PADDY (Hawking, 1990 and 1991) and FTR (Hawking and Bailey 1993) systems. These systems are discussed in more detail in the case studies section (2.6) below.

III). *Special parallel hardware.* A number of different special purpose parallel hardware architectures have been built for pattern matching in IR. The reader is referred to (Hollaar, 1992) and (Hurson et al, 1990) for more detailed information. One of the architectures, systolic arrays, can be classed under MISD. Systolic arrays (an example of cellular arrays) work by pattern matching characters every clock cycle in a pipeline where the target text and query travel in opposite directions. The associate memory architecture uses memory chips as the comparison devices, therefore patterns can be matched in parallel in actual memory. Finite State Automata (FSA) use transition tables over single cell comparator chips. The argument made for the use of these systems is that normal computing components are not very efficient at character comparison and therefore are not particularly good for pattern matching in text retrieval. Hence special purpose components are preferable to conventional computers since they offer a faster throughput for queries. This author does not agree with this argument. Inverted files have been shown to provide very efficient query service (at the cost of extra storage) for reasons which will become clear below. It is also very doubtful that these specially made chips could ever compete in price with general purpose chips: the cost of production and manufacturing CPUs is very expensive. We therefore do not see a future for these special purpose systems in IR. This is consistent with DeWitt & Gray's opinion that the future development of parallel systems will depend on standard components (DeWitt and Gray, 1992).

IV). *Networked Workstation Technology.* The current trend in parallel computing is to use a group of networked workstations or PCs, rather than special purpose machines. A great deal of interest has been generated in programming environments such as PVM (Sunderam, 1990) and standards such as MPI (Dongarra et al, 1996). One particular system discussed in this review (MARS) uses networked workstation technology for its hardware platform (Yount et al, 1991).The growth of distributed parallel processing has dealt a severe blow to many specialist parallel computer manufacturers such as Kendall Square Research, MassPar, Thinking Machines and AMT. Most have gone bankrupt apart from AMT which has metamorphosed CPP. DeWitt and Gray's opinion quoted above, on the development of parallel systems using standard components, is reinforced by the networked workstation technology factor. The trend towards workstation networks in parallelism has had significant impact on parallel IR systems (see section 2.6.4 below).

2.2.3 *I/O implications of different architectures*

One of the main qualities of IR is that in the main it is I/O bound rather than compute bound. This means that more time can be spent on reading in data from disk than actually doing computation. Thus a problem occurs where efficiency of the system is reduced because the data cannot be read in fast enough to service the computation. This problem is known as the I/O bottleneck and it is one that is shared with the area of Parallel database systems (DeWitt and Gray, 1992). In consequence, many of the systems mentioned above have very impressive I/O rates to overcome this I/O bottleneck. One architecture that addresses the issue is the shared nothing architecture described by DeWitt and Gray (1992). This architecture is classed under MIMD, and has a structure where CPUs have their own local disks to read data from. This reduces network traffic and disk contention considerably because data sharing is reduced to sub-sets of the whole data (which can be very large in database systems). Index maintenance costs can also be reduced. Tomasic and Garica-Molina (1992;1993a) make a very strong case for the use of shared nothing. We use the shared nothing architecture for experiments to be described in this thesis.

However such is not the whole picture. There are some IR computations which are compute bound and require considerable CPU resources. Examples are very large search spaces for passage retrieval and for query modification after relevance feedback, as used in Okapi at TREC experiments (Robertson et al, 1995). In such cases fast I/O cannot make much difference to the overall efficiency.

2.3 MOTIVATION FOR PARALLEL IR SYSTEMS

On the assumption that we want to do more and/or do it faster, there are two main reasons for using parallel computing in general. The first is that the speed of a processor is ultimately limited by the speed of light (Bell, 1992), when the maximum possible miniaturisation for components on a silicon chip has been achieved. The second is that the cost of placing silicon in smaller and smaller areas is very high in both the design and manufacturing of processors. The second limitation occurs long before the first and is therefore the major consideration.

The performance of IR systems is measured by the retrieval effectiveness and retrieval efficiency they provide (Frakes, 1992). Retrieval efficiency is the measure of the time taken by an IR system to do a computation on the database, although this usually means search it. The relative merits of the gain in retrieval efficiency by using parallel IR systems against their sequential counterparts can be measured by the speedup/efficiency and throughput measures defined in chapter 3. Users not only want fast and interactive access to documents, they also want to be presented with documents which are relevant to their needs; this is measured by the retrieval effectiveness of the IR system. The most commonly used measures for retrieval effectiveness are recall and precision (see chapter 3). Parallel IR systems have a place in providing retrieval efficiency for users and may well help in providing extra retrieval effectiveness.

The use of parallel computing specifically for IR has been quite controversial. Both Stone (1987) and Salton and Buckley (1988) have argued that an inverted file algorithm running on a sequential machine can outperform a signature file algorithm running on a parallel machine. The discussion in both papers originate from the work done by Stanfill and Kahle on the Seed system (Stanfill and Kahle, 1986). Since the Seed system uses surrogate coding (a response time of 2 minutes is stated for an example query), a sequential system using inverted files would in theory be able to offer a much faster response time to queries. This is because fewer comparisons and much less I/O is needed. Stone (1987) compares the performance of an inverted file on a single CM-2 node, while Salton and Buckley (1988) use the example of a Sun 3 to produce their theoretical results. Of the two studies Stone's goes much further. Stone put forward an alternative parallel algorithm to be used on inverted files in order to run the sequential inverted file system in a more efficient manner. Salton and Buckley (1988) are rather more negative and suggest that "the global vector matching systems developed over the past 25 years for serial computing devices appear more attractive in most existing text processing situations". It is hard to accept or reject this statement without knowing what they mean by most existing text processing situations, and without any analysis as to whether the global vector matching systems could also gain from parallelism. In response Stanfill, Thau and Waltz (1989) report an 80 fold performance advantage for a newer CM-2 against a Sun 4, which rather lessens the impact of the Salton and Buckley (1988) paper. Ultimately Stone has been proved to be correct, since most parallel IR systems use inverted files. The set merge on inverted lists can be computationally very intensive.

Four main reasons for applying parallel computers to IR have been suggested (Rasmussen, 1991): these are to improve response times, search larger databases, use superior algorithms and reduce search cost. We discuss each reason in turn below.

2.3.1 *Response times*

In situations where a large number of users need access to the system, a sequential IR system may not be able to offer the required performance of the application. In general when large numbers of users are logged on, the response time to the user is likely to be greatly increased. A related point is that of throughput: throughput is the number of queries or insertions which can pass through the system in a given time period. We may consider the use of query processing acceleration techniques with a resultant loss in retrieval effectiveness (Hawking, 1998) or parallel computing to improve response times. Parallel computation has the potential to offer faster response times for individual queries without loss in retrieval effectiveness and offers a greater throughput for queries and insertions as more memory resources are available. In extreme circumstances we may consider using both techniques to speed up processing. Response time is also dependant on database size in conjunction with multiple query service.

2.3.2 *Very large databases*

The response to user queries in very large databases (e.g. multiple Gigabyte) are likely to degrade particularly for those which have a reasonable rate of growth. In principle parallel systems tend to offer much better scaleup than sequential systems. Scaleup is defined in chapter 3. A query response time on a small IR system using a small database should be the same for a large IR system using a large database. It is important to introduce a note of caution as this point. This author does not believe that parallel computing can be usefully applied at this juncture to small databases with few or single user base (Stone, 1987). The emphasis in this review is very much on large scale text databases.

2.3.3 *Superior algorithms*

We stated in section 2.3.2 that we do not believe that parallel computing can be usefully applied to small text databases at this point. It may be the case at some time in the future that a given algorithm which requires more computation to complete its task will be able to offer a superior retrieval effectiveness performance in terms of precision and recall than previously implemented algorithms. For example there are a number of extended boolean models (Fox et al, 1992) which offer very good precision/recall at the cost of extra computation (which is high in the case of the P-NORM model because of the exponentiation operations required). Some in fact argue that extra computation will deliver much better results. Skillicorn (1995b) argues that regular expressions offer more powerful query capabilities than other searches. MacLeod and Robertson (1991) suggest that generally speaking the "most effective algorithms are among the least efficient".

There has been some debate on the merits of extra effort to achieve a better level of retrieval effectiveness. Blair and Maron (1985) evaluated a large operational full-text IR system over a six month period, and proposed a hypothesis in which deterioration of recall is a function of increasing database size. They argue that extra human effort at the indexing stage is needed to overcome this recall deterioration. Salton (1986) takes issue with their arguments, and they reply (1990). Although recall deterioration with file size must be regarded as unproven (and is particularly hard to prove empirically), the possibility both that it occurs and that it may be alleviated by more complex and more effective search algorithms is worth investigation.

A further issue arises from the very large search spaces which have been investigated in the Okapi at TREC research (Robertson et al, 1995). Because such a search space is so large it is unlikely that even parallel machinery will be able to explore all of it. A time complexity of $O(n^3)$ is reported for unoptimized passage retrieval (Robertson et al, 1995), where n is the number of text atoms. The search space for term selection on routing or filtering queries is so large no order value can be stated. Chapters 9 and 10 give results of our experiments on these search spaces using parallel computing methods by searching part of the overall space.

2.3.4 *Search cost*

Stanfill et al (1989) assert the cost effectiveness of an IR system is the ratio of database size to search cost, i.e. the resources used to search the database. Using the assumptions that database search is linear with the size of the database, speedup is linear for those algorithms which keep processors busy and resource costs (such as communication overheads) are static, Stanfill et al (1989) show that cost effectiveness asymptotically approaches a level of optimal cost effectiveness. By increasing the size of the database we can move the level of optimal cost effectiveness to a more favourable figure. It is stated that for a database of 1 Gigabyte the improvement in cost effectiveness is 100 fold, but for a 100 Gigabyte database the improvement is 10,000. However as Hawking (1991) points out a higher figure needs to be treated with caution because hardware (the CM-2) can only use a limited number of Data Vaults (the CM-2 storage system), which restricts the amount of text, let alone index information that can be stored. Hardware factors therefore limit the relevance of this cost effectiveness metric.

2.4. APPROACHES TO PARALLEL IR

In this section we describe approaches to parallel information retrieval using a classification due to Rasmussen (1992), influenced by Faloutsos's classification of access methods for text (Faloutsos, 1985). The classification does not differentiate between a particular algorithm and storage method. It is found that they tend to be bound together quite tightly in parallel IR systems. By algorithm we mean method of searching on the storage method and by storage method we mean organisation of the data on disk. The interaction between machine type and the classification is discussed in each of the sections below. The methods discussed are pattern matching, signatures/surrogate coding, two-phase search, inverted files, clustering, connectionist approaches and other miscellaneous approaches.

Some issues need to be addressed with respect to each of the algorithms in the classification. Firstly the assignment of tasks to processors will determine the level of performance gain over sequential systems: it cannot be taken for granted that using parallelism will automatically provide enhanced performance. The placement of tasks will also determine the level of interprocessor communication: an unavoidable overhead and one which may greatly degrade algorithm performance if data or task placement is mis-handled. Data distribution methods have a significant effect on task assignment and the subsequent level of interprocessor communication for a particular algorithm. Secondly there is the granularity of parallelism for an algorithm. Granularity can either be fine, coarse or mixed grain, meaning small, large or variable computation sizes: a computational unit being a single 'atom' of work which can be done by a processor. The type of query parallelism available in an algorithm is also very important: that is, the method of parallelism used to service user queries. Intra-query parallelism is parallelism within queries, that is a single query is distributed amongst processors. Inter-query parallelism is parallelism among queries, that is a number of queries are serviced concurrently. The concepts of data partitioning, granularity and query parallelism will be discussed with respect to each of the classes.

2.4.1 *Pattern matching*

Pattern matching is the method of searching the raw text in a given text corpus with a string query. There are a number of methods for matching patterns efficiently including the Knuth-Morris-Pratt and Boyer-Moore string searches (Wirth, 1986) and variations of these. In a system without parallelism, pattern matching normally involves the sequential scanning of every document in the system: no index is used. Methods include left hand truncation, variable length don't care (VLDC), a proposed implementation of the "computing as compression theory" SP algorithm (Wolff, 1994a), proximity searches and pre-computed patterns (Skillicorn, 1995a and 1995b). We describe below parallel methods which have been implemented or are proposed for pattern matching algorithms.

Using an example we can describe the operation of pattern match in parallel computing. Firstly we partition the target text among our processors. We then broadcast the whole pattern to all processors and the pattern is applied in parallel to each partition of the text. Results from the processors are sent back to the user for inspection. This scenario is rather simplistic, but it does give a flavour of the operation. A number of issues are thrown up by this example, in particular the operation of the algorithm on SIMD and MIMD architectures. The issue of how load balancing is affected by the implementation of the algorithm is important. With MIMD systems we can allow pattern matching on different processors to proceed independently of each other. The implementation on SIMD systems is slightly more problematic. Each pattern match needs to work in lock step on every processor: patterns may need to advance a computed distance. Unless we keep this computed distance in a local variable, a set of processors have to wait until others have 'caught up' in the computed distance and our load balance is reduced together with further loss in the efficiency within the chosen pattern matching algorithm. However with the computed distance we are likely to finish pattern matching on some processors, leading to a gradual reduction in processor efficiency as processors complete their tasks: this is a problem shared with MIMD systems. An alternative method for SIMD systems described by Pogue and Willett (1987a) is to broadcast individual characters to processors one by one which match them in that order. As each match is a made the presence of a hit is recorded, if and only if the previous character in the sequence was matched.

More complex patterns can be applied to text corpus in the same manner as the MIMD and Pogue & Willett algorithms. With MIMD we simply apply a utility such as grep or fgrep to each text partition on every processor in parallel. An example is the PADDY system (Hawking, 1992) which provides tools for the use of a regular expression library on each cell (processor) of a Fujitsu AP1000. An example of an algorithm implemented on SIMD systems which support complex patterns is variable length don't care (VLDC). For VLDC, prefix and suffix patterns are recorded: the presence of a word delimiter between the result set of prefixes and suffixes is then identified.

The SP pattern match (Wolff, 1994a) would use a completely different method. The SP algorithm works by broadcasting each character in the query, from left to right, to each character in the text corpus to make a true or false match. Given that it is impossible to have a processor for every character in the database, we can assume that each processor is given a set of characters. A tree structure is built up which records the probabilities of matches being useful, in decreasing order (matches nearer the root will have a higher probability of usefulness). The parallelism in the SP algorithm lies in the broadcast of characters and the ability to create and manipulate the tree structure for each text partition. A time complexity of O(Q) is claimed where Q is the size of the query pattern. It should be noted that the SP theory is controversial, and there has been heated argument as to the usefulness of it in practice (Wolff, 1994b; Stephen and Mather, 1994; Stephen, 1994). We are unable to comment on its usability in practical situations until an empirical study has been done using a parallel implementation of the SP search algorithm.

Hawking et al (1995) describes a method of parallel proximity searches on the PADRE system. A match set for each string in a query is created: this match set contains pointers to the first character of each instance of the pattern. Using some proximity value we merge these match sets by comparing the pointers and recording those pointers which meet the proximity value criteria. The set creation and merges can be done in parallel for each portion of text being searched in their respective cells. If documents are too large to fit in a single cell's memory, the cells need to communicate in order to complete the matching process: this inter-processor communication would reduce efficiency.

Skillicorn (1995a;1995b) describes a method of search which he asserts can be defined in terms of language recognition. The proposed algorithm uses a set of pre-computed patterns. Membership of textual data to these patterns is pre-computed in order to identify search patterns that have some common attributes. If membership of text to a pre-computed pattern is found it is placed in segments. The text would be partitioned across a given set of processors, the pre-computed pattern applied to the text and the search would access only those segments which are capable of matching a query. It is stated that where text is indexed as trees, regular expressions can be executed in logarithmic time complexity on a parallel computer.

An important theme in the algorithms described above is the distribution of text in one of two ways: either by text boundary (say documents) or by character (documents may reside over several processors). If text boundaries are crossed, more inter-processor communication is needed as processors need to exchange information. We can remove this problem by keeping documents as a whole in the processors. But this strategy itself has two main problems: the document may be too big to fit in a processor's main memory, and given that documents are likely to be of widely varying sizes a problem called data skew is observed. Data skew is caused when some processors complete their computations faster than others, which remain idle until the whole computation has finished. This can cause a loss of efficiency, in the worse case degrading the computation to that of sequential time complexity. Hawking (1994) has defined a measure of load imbalance LI in order to understand the effects of data skew on the pattern matching computation (this metric is defined formally in chapter 3). A method to overcome the problem is to try to arrange the documents in such a way as to reduce this LI value. A simple example of this is to place as many small documents on the same processor as possible. Where practical, large documents are placed in neighbouring processors to reduce inter-processor costs while smaller documents are used to fill up any extra memory. Breaking up the document into pages or paragraphs could also be useful.

The interaction between machine type and the classification tends to be based on the granularity of the computation. In the case of SIMD systems the granularity is that of a character, which is the finest grain that can possibly be used. With MIMD systems the granularity tends to be much coarser, but in fact is mixed granularity since documents are of varying sizes. The method supports intra-query parallelism and may be able to support inter-query parallelism with suitable processing, for example merging user queries and submitting them as a batch: we do not know of any systems which have implemented this.

The pattern matching algorithms are very search intensive, but they have a low storage cost and allow different types of searches such as left hand truncation which are difficult to implement in the algorithms classified below as the original text is needed.

2.4.2 *Signature / surrogate coding*

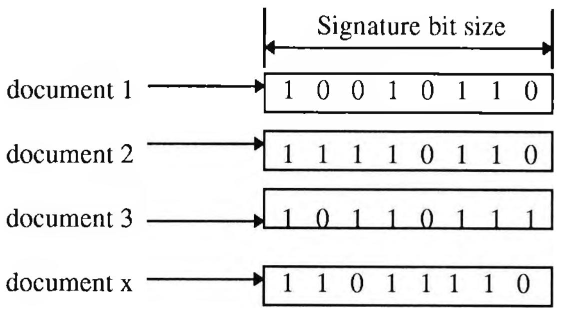
Text signatures are document surrogates which are generated by hashing terms on one or more bits of a fixed sized bit pattern (Faloutsos, 1985). Once these signatures have been generated they can be distributed to processors and searched in parallel. The search is done by applying the same hashing function to the query, as was applied to the documents. The search is therefore a fast bit comparison between the query and document surrogates. Pogue and Willett (1987a) describe an alternative method where integer values of the bit positions are broadcast one by one to the processors. The pioneering work described by Stanfill and Kahle on the Connection Machine has already been briefly mentioned in section 2.3 (Stanfill and Kahle,1986; Waltz, 1987). Other work includes a parallel Bit-Sliced Signature File (BSSF) method described by Panagopoulos and Faloutsos (1994) and Frame-Sliced Partitioned Parallel Signature Files described by Grandi et al (1992). Detailed descriptions of these different methods are given below.

Before we describe systems which use signatures it would be useful to review signature files (see fig 2-2). Signature file can be viewed as matrices where the rows represent document signatures and the columns represent individual bit positions of the signature across documents. We therefore have a number of partitioning methods for parallel computing on this matrix. The first, horizontal partitioning, represents row parallelism where signatures are compared in parallel (fig 2-2a). The second, vertical partitioning, represents column parallelism where sections of the signatures are compared rather than the whole (fig 2-2b). Vertical partitioning can be done across the collection or by a frame: a subset of the collection (fig 2-2c). A hybrid policy of vertical and horizontal partitioning can also be used. How these partitioning methods work in practice will become clearer in the discussion below.

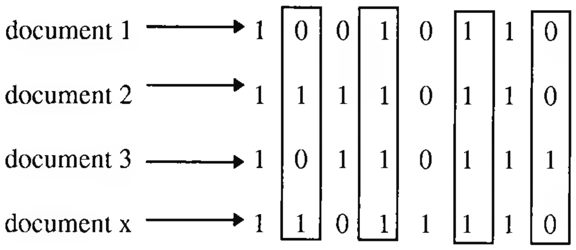
The Seed system described by Stanfill and Kahle (1986) uses the horizontal method (fig 2-2a) for partitioning the signatures. Seed uses a SIMD architecture, in this case the Connection Machine CM-2. The program works by loading signatures into memory, broadcasting the query signature to the processors to compare in parallel and retrieving the results. In theory it is possible to load a document signature in every processor, but Stanfill and Kahle assert that for a 512 bit signature "a limit of 15 to 30 words is reasonable". Therefore the system creates a number of signatures and spreads them across a number of processors if this upper limit on term to signature size is exceeded. Thus document sizes in the corpus have a direct effect on how many signature comparisons can be executed in a given search. The system allows the use of relevance feedback to reformulate a query. Reported results include a running time of 50ms for a 200 term query on a 112 Megabyte database. Estimates for a 15 Gigabtye database are also given with a running time of 2 minutes for a 25 term query and 3 minutes for a 20,000 term query. The latter estimates cast doubt on the usefulness of the system in interactive environments when very large databases are searched (compare these results with ours in chapter 7). This method of search has also been used in Transputer machines (Walden and Sere, 1989).

Panagopoulos and Faloutsos (1994) point out that the signature file for a very large database using horizontal partitioning may not fit in main memory, which has implications for their use in interactive applications: the Seed estimates given above bear out this argument.

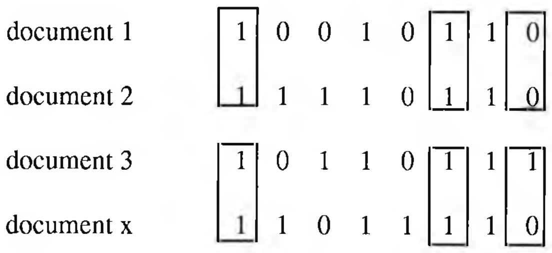
They therefore propose a Bit-Sliced Signature File (BSSF) which is based on vertical partitioning (fig 2-2b) on the bit level. The method would work by storing the signature file matrix by columns rather than rows. Each term in a query is hashed to a signature. The hashed positions of the query are identified and only those relevant column slices (or bit-slices) are fetched in to main memory and compared.



2-2a - Horizontal partitioning



2-2b - Vertical partitioning



2-2c - Frame partitioning

Fig 2-2. Forms of parallelism in signature files

A processor has a given number of bits with which to store a subset of the bit-slice. The algorithm would loop through these bits and compare subsets of the bit-slice in parallel. Where the bit-slices fit in main memory a total fetch policy can be used: where they do not, a partial fetch policy would be used, i.e. a subset of the bit-sliced identified from the query hashing. The proposed method would work on a SIMD architecture such as the Connection Machine CM-2.

Estimates for performance of the method include a response time of 2 seconds or less for databases up to the size of 128 Gigabytes using a CM-2 with 64K processors.

The work described by Grandi et al (1992) describes a hybrid method that combines both horizontal and vertical partitioning methods which they assert are suitable for implementation on parallel machines. The use of the shared nothing architecture described by DeWitt and Gray (1992) is recommended. The architecture of the system described is divided by three dimensions: frames (which are subsets of a signature), partitions (a horizontal fragment of frames) and blocks (a horizontal fragment of partitions). The signature file is stored in terms of the frames, each disk containing a subset of the frames (fig 2-2c). Thus frames are stored and can be searched in parallel while other frames are being serviced. Hence the classification of the method as being Frame-Sliced Partitioned parallel signature files. Since all frames would not be needed by a search, the method can allow inter-query parallelism as well as intra-query parallelism. While the method does overcome some of the limitations of those described above, this is at the cost of a great deal of extra complexity. This complexity in parallel systems should not be underestimated. Comparative results with the systems in this class are not available.

From the above discussion we can assert that the signature partitioning method interacts with the query parallelism directly allowable. Horizontal partitioning allows only intra query parallelism directly, while vertical partitioning and the hybrid method allow inter and intra query parallelism directly. Inter query parallelism could be supported indirectly if batch queries were used; although such would be problematic (see the discussion on false drops below). The granularity of signature files can be either signature, bit-slice or frame-slice and bit level granularity can also be used if the special hardware to work at that level is available.

The advantage of the method is that it is rather amenable to implementation on parallel computers. Since the signature matrix defined above has a regular shape we can reduce data skew quite considerably, although we may not be able to eliminate it completely given that signatures files may not fit into main memory. There is also a much lower storage overhead of about 10% compared with 50% to 300% found in inverted files (Faloutsos, 1985). However a serious drawback is associated with the method, the problem of false drops. Since different terms may hash to the same signature bits, collisions will often occur between query and document terms. A number of criticisms of the method have been made therefore in using the signature file method in an operational environment (Salton and Buckley, 1988), in particular that signatures cannot support sophisticated term weighting schemes. The subsequent effect of false drops on precision and recall can be profound. A further serious problem is that position information is lost, therefore proximity operations are unavailable in the class.

2.4.3 *Two-phase search*

This method has been proposed to overcome the high search cost of pattern matching and the low retrieval effectiveness of the signature method. The first phase of the search compares a signature version of the query with document signatures to create a hit list. The text arising from this hit list is then searched with the required patterns to eliminate the false drops and produce the final document result set. Since the number of documents pattern matched is greatly reduced, the increase in speed and effectiveness makes the method valuable. Parallelism can be used in both phases of the search. Two-phase searches have been implemented on SIMD machines at Sheffield University (Pogue and Willett, 1987a and 1987b; Carroll et al, 1988) and on a MIMD transputer network by Cringean et al (1988;1989;1990;1991a;1991b). Panagopoulos and Faloutsos (1994) also recommend the method's use when using signature files. Any of the signature and pattern matching methods described above could be used.

An example of the two-phase search can best be illustrated by looking at one particular system, the transputer network program described by Cringean et al (1988;1989;1990;1991a; 1991b). This system uses the process farm approach to parallelism to increase efficiency on the more computationally intensive second phase. The horizontal partitioning method is used for the first phase signature comparison. In this approach a single farmer distributes work to a number of worker processes who do the search. In the first phase the query signature is compared with document signatures (pre-loaded into memory) on a number of Transputers attached to the root transputer and a hit list of documents are recorded. In the second phase the farmer distributes the documents in the hit list to the workers, receiving the final document result set from them. A triple chain of Transputers was found to be the most effective topology. Data skew in the second phase is reduced since a worker is given more work on completion of a search: waiting for all workers to search a given set of documents would reduce the system's efficiency drastically. However it should be noted that documents may need to pass through several processors before reaching the target worker, because of the grid layout of transputer networks. The cost in extra communication and lost computation in routing processors affects the overall efficiency of the system. In the event this was found to be a significant problem: Cringean et al (1991b) state that a substantial increase in communication speeds would be needed for the method to achieve its full potential. A further interesting result was that a more efficient signature search on the first phase increased the amount of pattern matching needed in the second phase.

The granularity of two phase search is rather mixed depending on signatures' granularity in the first phase and documents in the second phase. Given that documents are irregular structures and signatures are regular, data skew is more prominent in the second phase of the search. The method supports intra-query parallelism for both phases. Inter-query parallelism however, could be used in the first phase if Frame-Sliced Partitioned Parallel Signature Files were used and for both phases if queries were submitted as batches. The interaction between machine type and the classification relates to the signature partitioning method for the first phase and computation granularity for the second phase.

2.4.4 *Inverted file*

Most commercial and academic IR systems use inverted files. The reason for this is that until recently query processing has been given priority over insertions, and inverted files provide much faster searches than other methods such as pattern matching and signatures. This is because the indexing eliminates the need for searches on many irrelevant terms. However the generation and maintenance of inverted files is very expensive and this makes its use problematic in applications where insertions are frequent. As stated in section 2.4.2 the storage requirements for inverted files are far costlier than any of the other methods reviewed in this chapter. In our description of the method below, we pay particular attention to the data partitioning schemes introduced in chapter 1.

The most prominent of parallel IR systems have used inversion as their storage technique (Reddaway, 1991; Stanfill et al, 1989; Aalsbersberg and Sijstermans, 1990; Stanfill and Thau, 1991; Stanfill, 1992; Linoff and Stanfill, 1993; Massand and Stanfill, 1994; Hawking, 1994). We briefly review the structure of an inverted file (Faloutsos, 1985): an index or dictionary file contains a list of keywords in the collection, number of documents in which that keyword occurs and a pointer to a document list: a postings file or inverted list contains the document list for all the keywords and may in some cases contain position information for each keyword in each document.

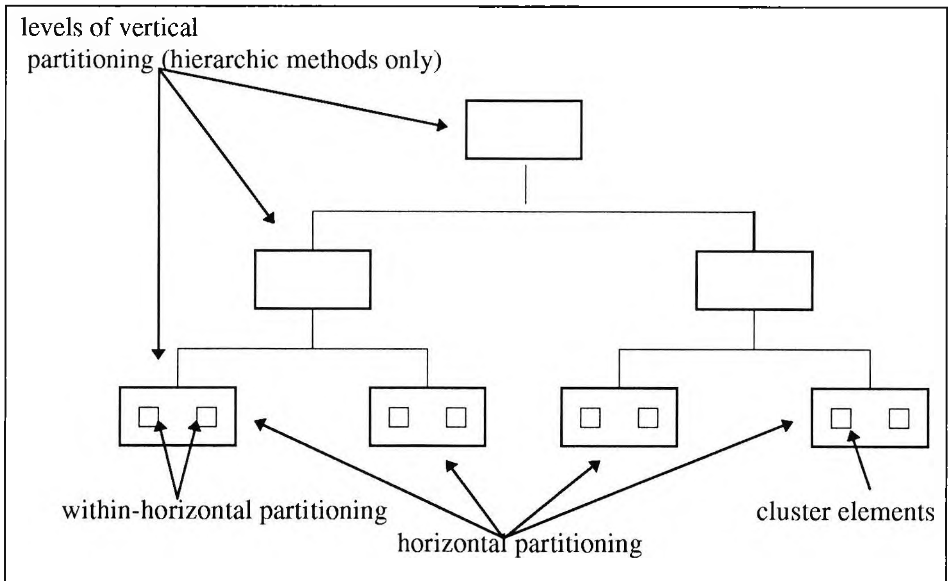
Jeong and Omiecinski (1995) discuss the effect partitioning in inverted files has on the performance of multiple disk systems. They advocate a shared everything approach as opposed to a shared nothing described in section 2.2.3. Multiple disks are used to exploit I/O parallelism. The use of a multiprocessor with shared memory is assumed. The two partitioning methods described in chapter 1 above are considered. The results produced by simulations are that *TermId* partitioning is best when the term distribution in the query is less skewed (or more uniform) and *DocId* partitioning is best when term distribution is more skewed (or less uniform). *DocId* partitioning sacrifices more I/O and space in order to ensure better load balancing in a more skewed query environment. When query term distribution is a little less skewed the postings for a term can be retrieved faster since disk access times for terms are more evenly distributed. When more skewed the load balancing of the machine will be affected by large disk access times for some terms. *DocId* partitioning avoids the latter problem by providing constant disk access times so that large access times for terms with very large postings are masked. This advantage is lost in a less skewed environment and the cost is greater because multiple disks have to be consulted in *DocId* partitioning (and the term accesses can be done in parallel). Inter-query parallelism is more difficult to service with *DocId* partitioning: each query term must take its turn on the disk queue. Term collection information is often needed for weighting calculations: this has an implication for the efficiency of term weighting using *DocId* partitioning (see section 2.5.3). Based on their simulations, Jeong and Omiecinski recommend that the Shared Everything architecture be used in medium sized Text Retrieval systems or as components in a larger shared nothing machine.

Tomasic and Garcia-Molina (1992;1993a) describe hybrid methods of partitioning inverted files on distributed shared nothing systems. They assume the existence of multiple disks per single CPU. They classify distribution methods as: Disk, I/O Bus, Host and System organisations. The Disk and System organisations are equivalent to *DocId* and *TermId* partitioning methods respectively. In the I/O bus organisation documents are distributed across I/O buses and inverted: this creates one inverted file per I/O bus. In the Host organisation documents are distributed to CPUs as per *DocId* partitioning, but the inversion is spread across the disks connected to the CPU. Where one I/O bus exists per CPU the I/O bus organisation is equivalent to the Host organisation. Simulations of full-text system and an abstract service were done using all the organisations described: in their results the Host organisation appeared to performance well for full-text systems, while the System organisation (or *TermId* partitioning) performed better on abstracts.

We can divide parallel systems which have implemented inverted files into two main camps, those which use *TermId* partitioning (Reddaway, 1991; Stanfill et al 1989; Ribeiro-Neto et al, 1999) and those which use *DocId* partitioning (Hawking, 1996; Aalbersberg & Sijstermans, 1990; Stanfill & Thau 1991; Hollaar, 1991). None of these groups have done comparative studies to ascertain which of the partitioning methods is best. Most of the research done on parallelism for inverted files has been done on the search task. Some work in the area of selective dissemination of information (SDI) has been done by Kapaleaswaran and Rajaraman (1990) using a subject division rather than data partitioning method.

The granularity of inverted files is based on the postings of the inverted list. Therefore granularity is much finer than the approaches described above (if we discount the possible use of special hardware to match at the bit level). One of the main reasons for the success of SIMD machines in parallel IR, is that they are very good at computing with this level of granularity. SIMD machines cannot normally handle inter query parallelism with inverted files, but a method of using several DAPs connected together has been put forward (Reddaway, 1991) which would overcome this limitation. Three systems which use inverted files are described in the case studies section (2.6) in more detail. More recent work at TREC can also be found in the case studies section.

2.4.5 *Clustering*



levels of vertical partitioning (hierarchic methods only)
within-horizontal partitioning
horizontal partitioning
cluster elements

Fig 2-3. Cluster parallelism

Clustering is a method of identifying similar documents, based on a given similarity method, e.g. distance in vector space using the cosine function. The documents are organised into groups or clusters, which in turn can consist of a single centroid and document vectors belonging to that cluster (Salton and Bergmark, 1981). There is parallelism in the clusters as well as between them: we term this horizontal and within-horizontal partitioning. Very fine grain parallelism (e.g. at the posting level) is also available within document vectors. A further issue is the type of cluster: they can be either hierarchic or non-hierarchic. Hierarchic methods introduce a further level of parallelism: we term this vertical partitioning. Fig 2-3 shows the forms of parallelism available in clustering. It should be noted that clusters can be overlapping and non-overlapping. We describe below parallel methods for generating and searching in the clustering method.

The generation of clusters are computationally very intensive: orders of $O(n^2)$ to $O(n^5)$ are not unknown. This makes their implementation on sequential machines problematic.

MacLeod and Robertson (1991) describe a neural network algorithm for document clustering using non-hierarchic methods. Neural networks are inherently parallel: Networks can be divided in layers and nodes within layers which allow parallelism in two directions. Parallelism is used in the MacLeod algorithm when each document vector is compared with the current set of clusters, iterating until a suitable cluster has been found or learned.

Rasmussen and Willett (1989) describe parallel computing for various hierarchic *agglomerative* clustering methods. Agglomerative clustering consists of building the tree bottom up. Hierarchical clustering can be represented by binary trees where nodes are clusters and the position in the binary tree represents the similarity measure between objects. Three algorithms are used for clustering; SLINK, Prim-Dijkstra and Ward. The SLINK algorithm only has parallelism in the calculation of the current row of the distance matrix. The Prim-Dijkstra algorithm is almost entirely parallel except for storage of link information. The Ward algorithm uses parallelism the on nearest neighbour method, i.e. identify a chain of related objects concurrently. The parallel SLINK algorithm performed less efficiently than its sequential counterpart, considerable slowdown figures being recorded. The parallel Prim-Dijkstra performed much better in relation to its sequential counterpart with speedups of 3.6 to 6.0 recorded on 4096 1-bit processing elements. The Ward speedups ranged from 2.9 to 4.0. They compared the results from an IBM 3083/BX3 mainframe against the ICL DAP and conclude that parallelism can provide significant speedups over serial systems in this type of clustering for large datasets.

While there are clearly defined partitioning methods for clustering, the arbitrary shapes of each of the levels will effect the search efficiency of the algorithm, e.g. clusters do not have the same number of document vectors or a hierarchy may not have regular binary tree like structure. Organising the clusters (and hierarchies where necessary) is therefore essential for the efficient search in this method. Frieder and Siegelmann (1993) formally argue that an optimal algorithm for assigning clusters to processors is NP complete and is therefore unusable. They propose a heuristic using genetic algorithms to address the problem. The algorithm terminates when either all document allocations are equal or after 1000 generations. Other researchers propose more conventional techniques.

Ozkarahan (1991) discusses search on non-hierarchic document clusters on the RAP.3 system. The clusters of document vectors and a centroid representing the vectors are distributed to a number of processors. A query vector is applied to the centroids, which if successful applies a second search to the document vectors in that cluster. While some regard this as useful, it is unlikely that the method would be able to compete in speed with inverted files. In any case the insertion of documents is likely to be prohibitively expensive. The RAP.3 system deviates from other systems in this review as the integrated multimedia application area is addressed.

Sharma (1989) describes a generic model for parallel IR using clustering techniques for both non-hierarchical and hierarchical methods. The hypercube topology is used together with dedicated disks for each node in the hypercube (i.e. shared nothing). The key is to distribute a subset of document clusters, to get the best load balance on search. Two schemes for partitioning clusters on a hypercube are described: one said to be for increasing retrieval efficiency and one of increasing retrieval effectiveness. In the efficiency algorithm closely related clusters are assigned to different sub-cubes such that the number of documents is equal in all sub-cubes. Within a sub-cube a cluster is spread across nodes, with the centroid assigned to one node. In the effectiveness algorithm clusters are recursively distributed across sub-cubes, each sub-cube has a smaller dimension than its parent. A hierarchical clustering algorithm is used, mapping the hierarchy to the hypercube. All levels of parallelism for clustering are used in these proposed schemes. The search consists of the broadcast of a query and the application of the query to the document database. In the efficiency algorithm the query is received at each node and comparisons are done concurrently. Similarity values for clusters (centroids) are collected and sorted and sent to a designated node which chooses the highest ranked clusters; these are requested from the relevant locations. In the effectiveness algorithm the query is received at each node and comparisons are done concurrently, similarity values at all levels of the hierarchy being calculated. The results are transmitted up the hierarchy and on this basis the highest ranked documents are chosen. The simulation shows that as cluster levels increase, response times in the efficiency scheme remain static, while in the effectiveness scheme time increases dramatically. In this case Amdahl's law (the asymptotic limit for the computation) hits the efficiency scheme at 128 processors and the effectiveness scheme at 1024 processors.

The granularity of the clustering approach can vary; either the cluster or the vector or even elements of a vector if an array processor such as the DAP is available. Both inter and intra query parallelism for search are available in the method. It is difficult to comment on the interaction between the machine type and the method, because of the multiplicity of clustering algorithms available. The arbitrary shape of the clustering algorithm determines the level of data skew and hence the search efficiency. Because of the expense of generating clusters, it is unlikely to be able to compete with inverted files: unless some benefit in retrieval efficiency can be demonstrated.

2.4.6 *Connectionist approaches*

These approaches use a network model to represent information in an IR system (Rasmussen, 1992). Many are related to the 'neural network' and 'spreading activation' areas of computation. They are inherently parallel, but extremely complex and poorly understood. Because of this their implementation on parallel computers is difficult and little work has been done in the area: research has concentrated on sequential implementations as a consequence (Kwok, 1989; Kwok and Grunfield, 1994). The MacLeod and Robertson algorithm (1991) described in section 2.4.5 which uses neural networks can also be placed concurrently in this class. It should be noted that these researchers take a very different approach to others described in this review. Because of the complexity of these methods we do not attempt to describe data partitioning, granularity or query parallelism for connectionist approaches.

One particular connectionist system is the PTHOMAS system described by Oddy and Balakrishnan (1991) and has been implemented on the Connection Machine. The theoretical idea behind PTHOMAS is to represent a holistic view of the documents and their relationships. The method uses a network structure of nodes (documents, authors, terms) where the arcs (edges) between these entities represent a relationship in the index and thesaurus. The network is a global graph representing the universe of the database. The user sees a context graph which is a subset of the global graph and is created by user action. Various component graphs may be discarded in the user interaction with the system. The algorithm used is computationally very intensive: a database with 10,000 document abstracts would create a network with 1 million nodes/edges. Oddy and Balakrishnan have not addressed the issue of how to implement these ideas/methods realistically for large collections and therefore we do not see the PTHOMAS as being a practical proposition for the foreseeable future.

2.4.7 *Other approaches*

There are a number of different approaches to parallel information retrieval which do not fit easily into the classes described above. We therefore describe below some other work, both practical and theoretical. These include vector processing, hybrid inversion, functional programming and relational database. Given the variety of approaches in this section we will not attempt to describe the interaction between architecture, the algorithms and the types of query parallelism used.

I). *Vector Processing*. Stewart and Willett (1987) describe an algorithm for nearest neighbour search using a multi-dimensional binary search tree, using networked microprocessors. Documents are represented by vectors, as is the query: the vector contains identifiers of terms in that document. Document collection is represented as a binary tree with the nodes associated with document term vectors (all nodes at the same level of the tree having the same vector) and the leaves having buckets with documents sets. Similar vectors are inserted in the left tree and dissimilar are inserted in the right tree. Query search is done in the same manner. An upperbound value is set and the algorithm backtracks using the value to find relevant buckets. The search is bounded by $O((\log N)^k)$ where k is a collection dependant constant. The level of k determines the amount of backtracking and hence the efficiency of the search. A special simulation language for the simulation of queuing systems was used to produce the results. Search is done by broadcasting a query down the tree, the answer being broadcast back up in the opposite direction: backtracking to nodes in the tree is done where necessary. The "Overlap co-efficient" is used as the similarity measure. The level of speedup deterioration was found to vary widely and was collection dependant.

Efraimidis et al (1995) describe a system called FIRE which uses a transputer based supercomputer to implement a parallel IR system based on the vector space model. They use an automatically constructed thesaurus based on a connected components evaluation algorithm. Their basic approach is either to keep the vectors in main memory or to load vectors in chunks, and then to compare them with a query using the cosine similarity function. There is no discussion of vector storage and insertion costs with respect static or dynamic text databases. An argument for their method could be that the insertion of a document vector to the end of a vector file is much less expensive than that of posting information to an inverted file and would therefore be good for dynamic text environments. They refer to Stone (1987) who discusses the offset of computation against storage and maintenance costs in detail, but without justifying the method of storing vectors separately, it is hard to see how the FIRE system avoids falling foul of his arguments. A sequential inverted file system may outperform their parallel vector processing system.

Some vector processing models use reduced dimensionality methods which focus on semantic structure of documents in order to increase retrieval effectiveness. In such cases a single document vector is used to represent documents as the methods do not keep keywords in their representation (hence inverted files cannot be used). These methods are highly parallel as processing of the query vector over document vectors is completely independent. Such a method is Latent Semantic Indexing (LSI) first proposed by Deerwester et al (1990). Letsche and Berry (1997) describe a system called LSI++ which uses a back end processor method to hide parallelism from the programmer. The backend system uses a master/slave process topology broadcasting the query vector to the processors and collecting results using a global sort. A collection of 24 SPARC5 machines were used, connected by fast Ethernet. Good speedup was demonstrated on a 100k record USENET collection reducing times from serial processing by nearly 180 times. A variation of LSI is DSIR (Rungsawang et al, 1999) which uses word co-occurrence to find document vectors which are close to queries vectors in a 'semantic space'. They use the same master/slave topology as Letsche and Berry (1997) using a network of Pentium processors implemented on the PVM system. Their results show problems with NFS bottlenecks when distributing document vectors and load imbalance from poorly distributed computations.

II). *Hybrid Inversion.* Yount et al (1991) describe the MARS system which they have implemented to store medical records. The system contains 850,000 medical records, 2.5 million medical references and 500 million indexed words. The system runs in a standard UNIX distributed environment, with the machines linked together by Ethernet. The system uses the shared nothing architecture. The MARS system uses many of the concepts and mechanisms of distributed systems such as threads, remote procedure calls (RPC), external data representation (XDR) and the client/server model, etc. Each text word is classed as an instance, and is stored in one of the archives which are distributed amongst servers residing on different machines. The instance (or posting) is a fundamental unit for locating and manipulating records. The instances have a segment id number (SID) to identify a host, a record id (RID) for a given record and word count (WC) to locate individual elements of a word in a record. The system uses a hybrid inversion method utilising a dynamically changing hash function to identify word to word id and inverse mappings.

III). *Functional Programming.* Deerwester et al (1990) describe an architecture which uses a server as an interpreter for a functional programming language that uses lazy evaluation. Clients can make requests to multiple servers, therefore the language can be evaluated in parallel. In particular the processing of inverted lists, which can in some cases be very large, is addressed. It is stated that without lazy evaluation of lists much extra computation is needed where examination of intermediate results suggests that processing of the lists is unnecessary. The implications for space complexity are also significant, where the intermediate results need to be stored. They state that functional programming is a useful way to implement the lazy evaluation of lists to prevent the extra time and space complexity which may occur with certain queries.

IV). *Relational Databases.* A great deal of research has been done on using parallel computing for relational databases (DeWitt and Gray, 1992). Experiments using parallel relational databases for have been reported at TREC-3 (Grossman et al, 1995) and TREC-4 (Grossman et al, 1996). The guiding principle of this work is that while parallel relational databases are common, parallel IR systems are rare. An inverted file structure is modelled using relations and keyword searches are done using SQL. The parallel database machine used is the AT&T DBC-1012 Model 4 (formerly Teradata). The I/O penalty of using relational databases in IR is addressed by using a query reduction technique based on term selectivity, which according to the results given do not affect precision and recall adversely. Clustered primary keys are used to reduce I/O even further, by placing inversion data on contiguous data pages. However, it is unlikely that parallel relational databases would be able to compete in speed with parallel IR systems because of the superior I/O performance of the latter. The I/O performance reduction occurs because the amount of space needed to store the index is dramatically increased. The increase is due to the way data has to be normalised in order to be placed in relations. Whereas inverted files may need only 1 disk access to read data for a term, relational databases will need to read in more data and also require more disk accesses.

2.4.8 *Summary of parallel IR approaches*

We have examined a wide variety of approaches which have had a varying degree of success in speeding up search using parallelism. For search it is difficult to see how, of all the techniques studied, the inverted file method can be surpassed with respect to response time for users. We therefore intend to use that technique in research described in this thesis. Much of the research concentrates on the search task with some discussion on the indexing and routing/filtering tasks. We believe it would be productive to look at a wider range of tasks and examine the issue of data distribution methods and performance. It is particularly useful to look at the index update task to see if parallelism can provide improvement in an area where inverted file technology has been weak.

2.5 RETRIEVAL MODELS USED IN PARALLEL IR SYSTEMS

Information retrieval systems use models in order to extract relevant information from text databases. The application of these different models can have an effect on both the retrieval effectiveness and efficiency of parallel IR systems, it is therefore important to consider them. We divide the models up into boolean, proximity, term weighting and regular expressions. They are discussed in turn below.

2.5.1 *Boolean model*

The boolean model is dominant in commercial IR systems, and most of the mainstream systems described in this review offer facilities for users to submit boolean queries. They have been implemented on systems such as the CM-2 (Stanfill and Kahle, 1986) using the signature method, the DAP (Reddaway, 1991) and POOMA (Aalbersberg and Sijstermans, 1990) machines using the inverted file method and PADRE (Hawking and Bailey, 1995) using the pattern matching method. PADRE allows union, intersection and difference operations on match sets, but these are equivalent to OR, AND and AND NOT boolean operations. The MARS (Yount et al, 1991) system also uses the boolean model as the basis for its query language. Parallel systems cannot improve the effectiveness of queries using this model, and depend on the user to generate effective queries. Naive users can find generating effective queries using the boolean model very difficult. Retrieval efficiency could be increased by parallelism, whether it be increase in speed on pattern match or fast set manipulation on inverted lists.

2.5.2 *Proximity models*

A very useful extension to the boolean model is proximity operations. They are used to find text atoms which are within a specified distance of each other, e.g. next to each other (adjacent), in the same sentence or within a given character distance. Among the systems which use proximity models are the DAP (Reddaway, 1991), pattern matching in PADRE (Hawking and Bailey, 1995) and MARS (Yount et al, 1991). The PADRE system provides the most detailed information on the proximity operations it allows. These include followed by (fby), not followed by (not fby) and a combined proximity/weight scheme called Z-mode (znear) (Hawking and Thistlewaite, 1996). The fby operation finds matches on terms which are within a given number of characters of each other. The not fby operation finds text in which terms are not within a given distance. The znear operation uses proximity spans to calculate relevance scores (we can therefore place this operation concurrently in term weighting models). As with boolean models, improvements in retrieval effectiveness using parallel computing are not found: but retrieval efficiency could be improved if overall efficiency is not reduced by extra interprocessor communication or load imbalance.

2.5.3 *Term weighting models*

One of the main methods used to improve retrieval effectiveness is to utilise one of the myriad term weighting schemes that are available. The dominant scheme had been the vector processing model with systems such as RAP.3 (Ozkarahan, 1991), DowQuest (Stanfill and Thau, 1991), Transputer Networks (Cringean et al, 1988; Efraimidis et al, 1995) and POOMA (Aalbersberg and Sijstermans, 1990), all using it in various forms. More recently variations of BM25 term weighting model have been used (Hawking et al, 1999), the technique we use throughout this thesis. PADRE (Hawking and Thistlewaite, 1996) offers a number of weighting schemes based on the inverse document frequency (IDF) measure. These models may use unnormalised term weighting (Hawking and Thistlewaite, 1996) or normalised (Hawking, 1992; Stanfill and Thau, 1991; Aalbersberg and Sijstermans, 1990; Efraimidis et al, 1995). Cringean et al (1988) do not specify the normalisation method. Others such as Reddaway (1991) and Jeong and Omiecinski (1995) do not specify a weighting scheme in their discussion of term scoring in their papers. A very important issue has a critical effect on the efficiency of a term weighting scheme on a parallel architecture: some schemes require collection information to calculate the weights. If this information does not reside in one place, i.e. a processor and its resident disk, the parallel machine needs to use interprocessor communication to merge the data held separately into a single figure. This bottleneck could affect the efficiency of the term weighting calculation. Many parallel machines provide facilities to do just this, e.g. the DAP (Bale et al, 1990) and Fujitsu AP1000 (Hawking, 1995) in the form of global network operations. Where this special hardware does not exist, the interprocessor communication may reduce efficiency. Unlike the two models discussed above, parallel implementation of term weighting may allow an improved level of retrieval effectiveness if the improvement in efficiency allows weighting methods to be used which are computationally more complex.

2.5.4 *Regular expressions*

Regular expressions give a user the ability to search for complex patterns in a single statement. They can be very computationally intensive and are best implemented on raw text. Examples of work using or proposing regular expression in pattern matching can be found in Pogue and Willett (1987), Hawking et al (1995) and Skillicorn (1995b). They can be undoubtedly very powerful in the hands of a very experienced user, but naive users may find them difficult to use effectively. Parallel computing could improve retrieval efficiency quite considerably, but we do not see how it could improve retrieval effectiveness.

2.6 CASE STUDIES - "STATE OF THE ART"

We present four case studies which are regarded as the most prominent of those discussed: two of them because they have been commercially successful and two because they are the most up to date systems or methods being used in research laboratories. Inverted file technology is used by all of the systems discussed in the case studies. Detailed information on the commercial systems is however limited. In our discussion in the case studies we describe the suitability of each system for the task, storage methods and granularity.

2.6.1 *DAPText*

DAPText (Reddaway, 1991) is a commercially used parallel text retrieval systems used by Reuters for their text retrieval purposes. The system uses a range of compression techniques on the posting lists for terms of varying hit rates. Those terms with the highest collection frequencies have postings represented in 8 bits and 16 bits, whilst 24 bits are used to represent rare terms. The higher the collection frequency for a term the more compact the compression method. Boolean operations on bit maps are reported to be very fast on the DAP. The main aim of the system is to provide very fast query processing on common terms, since merges on them are more computationally intensive than rarer terms. Position data is also held (in 12 bits), but is kept separately from the inverted list. The reasons for holding position data separately are for efficiency on queries which do not require position data and the variety of compression techniques used. Updates on the indexes are not done immediately: documents are added to a separate area of the DAP memory and merged with the main index data in a given timeframe. Processing of documents takes half a second for those of an unspecified average size. The DAP 610 can handle 35 boolean queries a second. Each query is handled one at a time, since SIMD machines do not allow separate threads of execution. Therefore no inter-query parallelism is possible, unless several DAPs are connected together.

Information about the system is limited. There is very little information on how keyword and inverted lists are manipulated. The system appears to offer a very fast search on the back of the compression techniques described. The granularity of the computation is determined therefore by the required compression method for a given term. There is no discussion on those terms whose distributions may hover between different compression methods, and the subsequent effect this may have on performance. More recent work on using hypertext and the DapText system is reported by Wilson (1996a;1996b).

2.6.2 *DowQuest*

The DowQuest system is also a commercially successful system. The Dow Jones News Retrieval Service uses the system for its Text Retrieval needs (Yount et al, 1991). The algorithms and data structures for the system are described by Stanfill & Thau (1991) and Stanfill (1990;1992). We outline some related work done by Thinking Machines which is described in Stanfill et al (1989) and Stanfill (1992): the contrast between the two algorithms is instructive. We also describe some further work done on an IR testbed for a more recent version of the Connection Machine.

	Processor 1	Processor 2	Processor 3	Processor P
Row 1	0 (0.1) *	1 (0.2) *	2 (0.3) *	3 (0.4)
Row 2	4 (0.9)	5 (0.2)	6 (0.6)	7 (0.6)
Row 3	8 (0.7)	9 (0.5)	10 (0.8)	11 (0.4)
Row n	12 (0.6)	13 (0.9)	14 (0.1) *	15 (0.7) *

Relevant postings for a term are 3 to 13: * signifies irrelevant postings
(weights for postings are in brackets)

Fig 2-4. Assignment of postings to processors

The algorithm described in Stanfill et al (1989) works by multiplying a query weight with stored postings weights in parallel and sending the result to a mailbox somewhere on the machine. The *TermId* partitioning method is used. Using a data map (the keyword index), rows of postings are identified and placed in memory ready for computation. Processors are given an equal number of postings (n). The algorithm then iterates through each posting row of the processor, i.e. from row 1 to n, calculating weights for terms if and only if the posting in that row is identified as being relevant: otherwise the processor is deactivated (see Fig 2-4). The weights are then routed to the relevant mailbox in the machine after an iteration using a send and add command. When weights have been computed, the top documents are identified by sorting the weights in the mailboxes. This mailbox algorithm has been criticised by Reddaway (1991) who points out that term distribution will have an impact on its efficiency. If the postings lists are too large to be fitted in the machine at one go, the remaining postings can be loaded in from disk and processing can start again from row 0. SIMD machines are very good at this kind of fine-grain computing. However the algorithm suffers from a data skew problem when a row of postings only has a small number of active processors, e.g. one or two in a 64k processor machine. The effect on efficiency can be drastic, reducing the complexity to that of sequential machines in the worst case. To address this problem, partitioned posting file methods are discussed (Stanfill and Thau, 1991).

The partitioned posting file method described in Stanfill & Thau (1991), does not eliminate data skew but does reduce it considerably. Essentially postings are partitioned such that all term postings for a document are handled by a single machine node: thus the *DocId* partitioning method is used. This eliminates the need for the routing process for the mailbox algorithm. Postings are placed into blocks of a partition. The data map is used to identify the required partitions. The partitions are then loaded into memory and computed in parallel. The algorithm iterates through the partitions until a weight for every hit document has been calculated. The granularity of the computation is still the posting. Extra space is added to the postings file in order to retain alignment as far as possible. As with the DAP the system would appear to offer very fast search facilities. DowQuest was written for the CM-2 version of the Connection Machine. A prototype (Massand and Stanfill, 1994; Linoff and Stanfill, 1993) was written for the Connection Machine CM-5: a more powerful machine with a hybrid SIMD/MIMD architecture.

Massand and Stanfill (1994) and Linoff and Stanfill (1993) describe methods and data structures implemented on an IR testbed for the CM-5. They take the standard boolean model and extended it with proximity operators. Techniques for distributed databases are considered in particular the problem of term weighting across distributed collections. Compression methods are used to reduce the size of the inverted file: compression is applied to position data, but not to weighting data. They claim the decreased time in I/O can fully compensate for the decompress computation (based on a study of two corpora; the King James bible which is 4.5 Mbytes and a sample of Wall St journal articles which is 12.3 Mbytes). The issue of updates is considered as well as deletes: they use an in-core technique for the text database using the *DocId* partitioning method for inversion. Fixed sized blocks are used to distribute documents and re-adjust to text boundary accordingly (each processor looks after its own document set). A two pass index algorithm is used: the first pass calculates the space needed for each inverted list and the second pass indexes the text and puts inverted data into pre-allocated blocks. This algorithm took 20 minutes in comparison with the 90 or so seconds on the Fujitsu AP1000 reported by Hawking running the PADRE system (Hawking, 1995). Part of the differential could be the cost of compression, and part in having to do the indexing twice. In the event the prototype or test-bed did not become a product.

2.6.3 *PADRE*

More information is available on PADRE and its precursors than any other system covered in this chapter, and the system continues to be used for research purposes. We have already imparted much information on the system ranging from the hardware it uses (section 2.2.2), methods of operation (section 2.4.1 and 2.4.4) and query models available for the system (section 2.5). We therefore restrict our discussion to the history and philosophy of PADRE.

The system started life as PADDY (Hawking, 1991 and 1992) and concentrated on linguistic and lexicographic research on the *Concise Oxford English Dictionary* structured in SGML. Searches are based on the PAT indexing method (Gonnet et al, 1992), to implement pattern match, proximity and regular expression operations. Results from searches on the indexes show speedups ranging from 30 to 1000, where the speed of indexed matching depends largely on hit matches (Hawking, 1991). There is much discussion on the time to load data, a problem overcome by the introduction of the HiDIOS file system. A vision of the libraries of the future is given by Hawking (1992) who argues that a number of advantages lie with using parallel supercomputers including: libraries would be open for much longer, a number of people could read the same book, books are never lost or mis-shelved, catalogues are never out of data etc. He does however point out that there may be many practical reasons, such as legal and financial, which may prevent the complete replacement of libraries by parallel supercomputers.

The *ftr* system (Hawking, 1993) builds on work done in PADDY and while retaining its capabilities is oriented towards more conventional IR problems such as retrieving text. A user interface called *retrieve* is introduced in order to provide a more user friendly access to the applications services, rather than a command line interpreter (although this is still available in *ftr*). A significant decrease in load times is recorded for *ftr* over PADDY. The system also has the ability to load more than one text database.

The PADRE system retains many of the features of both *ftr* and PADDY, while introducing others such as inversion of text (Hawking, 1995), term weighting (Hawking, 1994), natural language processing techniques (Hawking and Thistlewaite, 1995), multiple user facilities (Hawking et al, 1995) and proximity spans (z-mode) (Hawking and Thistlewaite, 1996). The *DocId* partitioning method is used with partitioned indexes and postings. The reasoning behind the partitioning method is to provide fast update on inverted files while providing fast responses to user queries. Near linear speedups for indexing are reported. The searches on indexes are reported as being constant, whereas the search time for pattern matches decreases with increase in the number of AP1000 cells. Responsiveness to additions and deletions to a text corpus are recorded. Using 509 Mbytes from the Wall Street journal and 10 Mbytes of Associated Press reports a merge time of 18.7 seconds, of which half was the approximate load time from the host. A time of 9.2 seconds is reported for the deletion of all documents with the word 'computer' in them: this reduced the Wall Street journal collection by 57 Mbytes. The implementation of time-outs on searches (Hawking et al, 1995) is recommended to ensure reasonable responses times for users and to avoid 'killer queries' which can greatly reduce system throughput. More recently PADRE has moved to the cluster computing model (see below).

2.6.4 *Cluster Computing*

It is clear from recent research that standard components as part of networks of workstations is now dominant in the field of parallelism in IR. Cluster computing has revived the field of parallelism for IR after a three or four year moratorium. Cluster computing has been used within the framework of NIST's Text Retrieval Conference (TREC) series to examine performance over very large databases: initially over a 20 Gigabytes collection (Hawking and Thistlewaite, 1998) and then a very large 100 Gigabyte web collection (Hawking et al, 1999; Hawking et al, 2000). Participants have used a variety of architectures such as DEC Alpha, SGI, Intel and Sun, while using processor sets ranging from just 2 up to 20. Most of these systems split the collection and place the index on local disk using the *DocId* partitioning method. The granularity of parallelism used in cluster computing is very coarse grain. One group (ANU) tried using a RAID disk with subsequent performance degradation (Hawking et al, 1998). Some use the shared nothing architecture while others used a shared memory machine configuration. It is difficult to make any comment on which of these architectures are best as participants in these TREC experiments use very different methods from each other. We give our results in the Web Track experiments throughout the thesis (MacFarlane et al, 2000a).

We also contacted various web search engines to find out what type of parallelism they use. Understandably they were very reticent about giving out information on the methods and architectures they use and few responded to my enquires. The most helpful was Dixon (2000) who informed us that Google uses 4,000 Intel Pentium PIII boxes in three server farms which in their words are used to "index, categorise and prioritise the Web and return results to searchers". These Linux servers are stripped down and customised for Google's requirements. Altavista used to use 16 nodes each with 10 TurboLazer Alpha processors together with 200 Gigabytes of disk space and 8 Gigabytes of in-core memory. They have recently moved to using Compaq equipment, but were not willing to describe either the configuration or the architecture they now use (Shissler, 2000). Northern Light (Kim, 2000) were only able to give brief details on the ranking, NLP and clustering methods they use, which in itself was interesting but not very useful for our purposes.

2.7 SUMMARY AND CONCLUSION

This chapter gives an overview of the application of parallel computing to IR systems. We describe a classification much used in parallelism and describe some of the architectures which have been used to implement parallel IR systems. Issues such as the implication of I/O on different architectures are discussed. We describe a classification of approaches to IR due to Rasmussen (Rasmussen, 1992) which includes pattern matching, signature/surrogate coding, two phase search, inverted file, clustering and connectionist approaches. The importance of such issues as data partitioning and data skew are stressed in the discussion of each class. Other approaches such as parallel relational databases are also described. We describe the motivation for using parallel computing in IR as being good response times for users providing added retrieval efficiency, scaleup and machine efficiency on very large databases, allow for the use of superior algorithms (which provide a higher level of retrieval effectiveness) and lower search cost. In contrast we do not believe that parallel computing can be usefully applied at present to small databases with a small user base. The retrieval models used in parallel IR systems such as boolean, proximity, term weighting and regular expressions are described as is the impact of parallel computing on the retrieval effectiveness and efficiency of the models. The case study section gives detailed information on the DAPText, DowQuest, PADRE and parallel IR systems plus recent work in Cluster computing for IR. For further information on many of the systems described in this chapter, the reader is referred to Rasmussen (1991) a special issue on parallel processing in IR as well as Willett and Rasmussen (1990) for a large body of work done on the DAP.

Much of the work described above focuses on searching of text using various parallel methods, but there has been little focus on indexing text, using passage retrieval techniques, index maintenance or routing/filtering. In particular there is no overall survey of data distribution techniques for inverted files for all of the tasks being considered. The purpose of this thesis is provide this overall survey and to find out which data distribution method is best overall and for a particular task. Many of the algorithms described above are specifically written for one particular parallel machine and would be difficult to port. We aim to provide a system which is portable across machines.

Chapter 3

Methods, Data and Metrics Used

3.1 INTRODUCTION

In this chapter we declare various methods, data and metrics used throughout the thesis. Section 3.2 describes methods such as the retrieval model, index model, architecture and parallel machines used. The data used in our experiments is described in section 3.3 while section 3.4 outlines metrics used to measure both retrieval effectiveness and retrieval efficiency in the thesis. The software developed for the project is discussed in chapter 4.

3.2 METHODS

$$CW(i,j) = \frac{CFW(i) * TF(i,j) * K1+1}{K1 * ((1-B)+(B*(NDL(j)))) + TF(i,j)}$$

Variables

CFW(i,j)	: Collection frequency weight log(N) - log(n).
n(i)	: The number of documents term t(i) occurs in.
N	: The number of documents in the collection.
TF(i,j)	: The number of occurrences of term t(i) in document d(j). (term frequency)
DL(j)	: The total number of terms in document d(j)
NDL(j)	: Normalised document length (DL(j) / average DL for all documents.)

Constants

K1	: Constant that modifies influence of term frequency.
B	: Constant that modifies effect of document length.

Fig 3-1. The BM25 term weighting function used in the experiments

3.2.1 *The Robertson/Sparck Jones Probabilistic Model*

In chapter 2 we described the term weighting schemes that have been used in parallel IR systems. A notable exception until recently has been the use of the Robertson/Spark Jones probabilistic model (Robertson and Sparck Jones, 1976;1994), which has become increasingly influential in recent years particularly within the TREC conference framework. The model provides a theoretical framework for weighting terms based on the probability of relevance. In all our experiments we use the BM25 term weighting function (see figure 3-1), which uses statistics from the collection to rank documents in order of probability of their relevance to the user: term weights are not normalised. Two constants are used to modify the influence of various aspects of the weighting function, allowing BM25 to be optimised to a given collection.

Much of our effort in improving the retrieval effectiveness has concentrated on tuning constant variation, particularly in our TREC-8 experiments (MacFarlane et al, 2000a) which we summarize here. Varying tuning constants is an easy way of improving effectiveness and very little effort is needed in order to conduct experiments. There are two constants defined for BM25 (see Fig 3-1): **K1** that affects the influence of term frequency while the constant **B** is used to modify the effect of document length. Given that there has been no systematic work done with Okapi we also decided to examine the relationship between the two constants as well as the relationship between those constants and other variables such as recall and precision (see below for the definition of these). We hypothesized that optimum constant values for one data set would give good effectiveness in another data set (where a data set is defined as a document collection and query set pair). We found the hypothesis was validated only if the collection was the same in both data sets.

3.2.2 *The Shared Nothing Architecture*

We referred to the shared nothing architecture in our review of parallel computing in IR. For most of our experiments this is the architectural method we use for running jobs. We have used three sets of machines or clusters in this class to do our experiments: an eight processor Alpha Farm, a 12 processor Fujitsu AP3000 and a 16 node Pentium cluster named "The Cambridge Cluster". For the Alpha farm, each node is a series 600 266Mhz Digital Alpha workstation with 128 Mbytes of memory running the Digital UNIX 4.0b operating system. One of the nodes has a RAID disk array attached to it and other nodes can access the RAID using NFS. Two types of network interconnects were used: a 155 Mbytes/s ATM LAN with a Digital GIGASwitch and a 10 Mb/s Ethernet LAN: most of the indexing was done on ATM. The AP3000 consists of 12 Ultra1 nodes, each with 125Mb of memory, with a top rate of 200Mb per second network interconnect. Only 8 of these nodes were available to us in a single partition. Each AP3000 node has its own local disk with a processor speed of 167 Mhz. The "Cambridge Cluster" consists of 16 Duel processor Pentium PII nodes each with 384 Mb of memory and 9 Gb of disk space. The nodes are connected with by a fast Ethernet switch and a

Myrinet Gigabit class switch and have a speed of 300 Mhz per processor. The Alpha farm and AP3000 machines are located at the Australian National University in Canberra while the "Cambridge Cluster" is located in Microsoft Research in Cambridge.

3.2.3 *Other Architectures used*

We also used another architecture for our routing/filtering experiments. This model is made up of a master with a single disk that broadcasts the data required to each slave node. Each node has its own memory (the distributed memory MIMD architecture described in chapter 2). This included the Fujitsu AP1000's at Imperial College, London and the Australian National University, Canberra, and a network of heterogeneous Sun workstations at City University The AP1000 is a distributed memory MIMD (Multiple Instruction, Multiple Data) with a 2-D torus topology (Fujitsu, 1994). Both the ANU and Imperial machines have 128 SPARC1+ processors referred to as cells each with 16 Mb of memory (with a total memory of 2 Gigabytes), but variations of the AP1000 can have as many as 1024 cells each with sixty-four Mbytes of memory. The SPARC1+ processor speed is 25 Mhz. Data can be moved via a 2 Mbyte per second link, but a FDDI interface is capable of nearly 10 Mbytes per second for data movement. The network of Suns includes Ultra's, SS10's, SS20's and Sparc5 machines operating over a congested Ethernet cable, with a maximum possible bandwidth of 10 Mbits per second.

3.2.4 *Index model*

We used a variation of the traditional inverted file (Harman et al, 1992) in order to index documents for the various tasks in our thesis. We used a clear keyword and postings file split, often referred to as the dictionary file and inverted list respectively. The dictionary file consists of records with the following structure: keyword, collection frequency and pointer to the inverted list. Each element of the inverted list stores the PLIERS document identifier together with the term frequency. We are able to include positional data as part of our posting lists that has the following structure: field, paragraph within field, sentence within paragraph, word position within sentence and number of preceding stop words. This data was needed for the passage retrieval task and could be used for adjacency operations. We store the position information contiguously with each posting in the inverted list. Also included in the index was a document map file that was used to store the PLIERS and TREC document identifiers, the address of the document on disk, the document length and a list of passages with their lengths if position data is stored in the inverted list. The map is needed to provide a cross reference between PLIERS and TREC document identifiers, and to provide data for the BM25 term weighting function both in normal term weighting search and passage retrieval.

3.3 DATA

3.3.1 *Web Data*

For our indexing, search, passage retrieval and update task experiments we used the VLC2 (Hawking et al, 1999) or WT100g (Hawking et al, 2000) data collection as it has more recently become known. Table 3-1 gives the details of the collections used.

COLLECTION	**WT100g or VLC2**	**BASE10**	**BASE1**
No Documents	18,500k	1,870k	187k
Text Size in GB	100	10	1
Collection Word Length (Million)	8,600	865	87
Description	Full Db	10% of WT100g/VLC2	1% of WT100g/VLC2

Table 3-1. Web data collection details

The WT100g/VLC2 collection consists of 100 GB spidered web data. The BASE1 and BASE10 are baseline collections of the WT100g. We used various query sets over these collections for the search, passage retrieval and update tasks that are declared in the relevant chapters below. We were only able to use the full 100GB collection when doing our TREC-8 experiments. We also use subsets of the BASE10 collection in our indexing experiments that we name BASE2, BASE4, BASE6 and BASE8. The subsets were created by varying the number of BASE10 compressed text files put through the indexing mechanism (130 files per node for BASE2, 260 for BASE4, 390 for BASE6, 520 for BASE8). Each of the BASE **x** collections is approximately **x** Gigabytes in size e.g. BASE6 is approximately 6 Gigabytes in size.

3.3.2 *Routing/Filtering Data*

The databases used for the Routing/Filtering experiments was the Ziff-Davis collection and the TREC8 routing collection (routing/filtering concepts are defined in section 1.4.5). We describe each of these databases used and how they were handled below.

The Ziff-Davis collection is from the TREC-4 disk 3 and is 349mb in size with a total number of 161,021 records (Harman, 1996). Of all the accumulated (and somewhat complex)

TREC material sets of documents and topics, we felt that a collection of this size was the best for our needs given the restrictions on resources available at the time. Three databases were created for the Ziff-Davis collection: an extraction database, a selection database and a comparison database and used as per the Okapi methodology (Robertson et al, 1995). The databases were created by assigning documents to the three databases using a round robin method, i.e. document 1 to the extraction database, document 2 to the selection database, etc. Note that the extraction and selection databases form the training set, while the compare database is the test set. The training set is split into two in order to reduce overfitting when optimising queries using term selection. The total inverted file size for all three databases was 83mb (24% of the text file size). Each database was about 25mb in size. No position information was saved in the posting files (such data is not used in term selection). A stop word list compiled by Fox (Fox, 1990) of 451 words was not indexed. Words were conflated using a Porter stemming algorithm. There were 19 TREC-4 topics used for routing on the Ziff-Davis database, they are; 54, 57, 63, 65, 66, 75, 94, 95, 96, 97, 98, 100, 109, 113, 117, 128, 136, 223, 248. These were chosen on the basis of the number of relevance judgements available for routing/filtering: it was felt that topics with too few relevance judgements (i.e. one or two) would not be much use in the second level selection process due to excessive overfitting. The distribution of relevance judgements for the database was as follows; 1868 (39%) for extract, 1469 (30%) for select and 1483 (31%) for compare.

DATABASE	**Batch Filtering**	**Routing Training 1: EXTRACT**	**Routing Training 2: SELECT**	**Test Collection**
No Documents	64,139	251,396	256,761	140,651
Text Size in MB	167	979	1,013	382
Index Size in MB (% of Text)	37.72 (22.3%)	158 (16%)	162 (16%)	85 (22%)
Relevance Judgements (Avg)	548 (10.96)	1836 (36)	1797 (35)	1276 (25.52)
Collection Word Length (Million)	27	138	142	60
Description	FT 1992: Disk 4	1/2 of Disk4/5 Minus FT1993/4	1/2 of Disk4/5 Minus FT1993/4	FT1993/4: Disk 4

Table 3-2. Filtering track data collection details

We were able to gain access to much larger resources for a brief period and participated in the TREC8 routing track (MacFarlane et al, 2000a). Details of databases used for batch filtering and routing sub-tasks are give in table 3-2. The TREC8 database is much larger than Ziff-Davis and it was handled differently according to the TREC8 routing and batch filtering sub-track needs. That is, two separate databases were defined by the sub-track, one training set and one test set, both derived from disk 4 and 5 of the Tipster collection. The training set for the routing sub-task consisted of 500K documents which was split as equally as possible to create the extract and select databases. We did all the training for the batch filtering sub-track on the 64k record training set because of the limited number of relevance judgements for that data. The training set consisted all of tipster disk 4/5 minus the FT1993/94 data. The TREC8 data was treated the same as Ziff-Davis, apart from the stemming in which the Lovins method was used. The topics used were 351-400.

3.3.3 *TREC8 AD-HOC DATA*

We also used the TREC8 AD-HOC collection for the passage retrieval task. This text collection consists of Tipster Disk4 and Disk5 minus the Congressional record on disk4. It consists of 528,155 documents and the text size is 1,904 MB: the total word length detected was just under 270 million words. The topics used were 401-450.

3.4 METRICS USED

There are a number of different performance measurements or metrics for both retrieval effectiveness and efficiency experiments. We declare the ones we have used for the purposes of this thesis here.

3.4.1 *Retrieval Efficiency Metrics*

3.4.1.1 Elapsed Time

Elapsed time is the most important metric to be used in this thesis. Many of the other parallel metrics described below are in some way reliant on this one. By elapsed time we mean the time it takes a job from initiation to completion: this requires access to some time library functions such as *gettimeofday*. This metric is considered to be a 'black box' performance measure. For elapsed time on those tasks that would be used in interactive environments, we try and meet the 10 second criterion for searches suggested by Frakes (1992). This criterion would obviously not be applied to batch processing tasks such as indexing and term selection for routing/filtering which may take many hours to completed. Time may be recorded in milliseconds, seconds or hours as seems appropriate to the task being examined. As well as elapsed times we record the overheads in a parallel system which would not be a part of any sequential program's run time. We may when necessary state the percentage for total time for some aspect of a task where it is deemed significant element of overall time. Note that these measures were taken when the machine was dedicated to servicing the relevant IR task.

3.4.1.2 Extra Cost Ratio for Position Data

We stated in section 3.2.3 above that we are able to store position data in our inverted lists. Clearly this is an extra cost not only in space, but in extra time as well (particularly I/O time). In order to examine this extra cost we use a ratio on the elapsed time defined in fig 3-2. This metric only applies to runs where indexes of both types have been generated for the same text collection. The ratio allows us to quantify the extra cost in time for storing position data, for situations where different types of queries are submitted to a system (some of which may not need position data for processing). Ideally we would want the ratio to be as close to 1.0 as possible: this is dependant on the size and storage method of position data.

$$\text{Ratio} = \frac{\text{Elapsed Time for a given task on an index with position data}}{\text{Elapsed Time for the same task on an index with postings only}}$$

Fig 3-2. Extra Cost ratio in time for keeping position data in inverted list

3.4.1.3 Throughput

Throughput is a measure of how much work can be done in a given time period, and can be used for comparison purposes. It may refer to a single job (such as indexing) or several (processing of multiple transactions). Throughput measurement varies with the task, e.g. transactions and queries per hour, Mbytes of text processed per hour and evaluations per second.

3.4.1.4 Speedup

$$\text{Speedup} = \frac{\text{Time for task on sequential machine}}{\text{Time for the same task on a parallel machine with n nodes}}$$

Fig 3-3. Speedup calculation: speed advantage of parallelism

Speedup is a measure of the gain in speed over sequential machines and is calculated by dividing the time spent on computation using the sequential machine by the time using the parallel machine (see fig 3-3). A speedup which equals the number of nodes is said to be linear, greater than the number of nodes is said to be super-linear. A figure that is below one is said to be a slowdown (using parallelism brings a disadvantage to the chosen task). Whilst speedup is the accepted way of examining the performance of parallel systems, it should be noted that its usefulness has been brought into question (Hockney, 1993). For example you cannot use speedup to compare two algorithms as speedup for algorithm A say can be better than for algorithm B, but algorithm B may record a faster elapsed time. We therefore only ever use this measure within the context of one algorithm on one machine type.

3.4.1.5 Efficiency

$$\text{Efficiency} = \frac{\text{Speedup on N nodes}}{\text{N}}$$

Fig 3-4. Efficiency calculation: effectiveness of parallel machine use

Efficiency gives a measure of how well a particular algorithm scales when nodes are added (see fig 3-4). The metric also measures how well nodes are utilised in a parallel system of a given size. It is found by dividing the speedup found by the number of nodes used. An efficiency of 1.0 is desirable, but rarely if ever achieved. The aim is to achieve a near 1.0 efficiency result. The caveat expressed on speedup above also applies to this metric.

3.4.1.6 Scalability

$$\text{Scalability} = \frac{\text{Average Task Elapsed Time (Smaller Collection)}}{\text{Average Task Elapsed\_Time (Larger Collection)}} * \frac{\text{Data Size (Larger Collection)}}{\text{Data Size (Smaller Collection)}}$$

Fig 3-5. Scalability measurement

We define a metric of scalablity that is a measure of how well the algorithm scales on the same equipment (see fig 3-5). By the same equipment we mean the same parallel machine. The measure is the proportion of time as measured against the database or collection size. This metric has the advantage over simple ratios in that its result actually relates task processing times to the size of data in question. A figure of 1.0 gives linear Scalability, less than 1.0 means there is a loss while greater than 1.0 means the gain is super-linear.

3.4.1.7 Scaleup

Scaleup is defined by DeWitt and Gray (1992) as "the ability of an N-times larger system to perform an N-times larger job in the same elapsed time as the original system" (see fig 3-6 for a more formal definition). A scaleup of 1.0 is said to be linear, less than 1.0 means there is a reduction is scaleup while greater than 1.0 means the gain is super-linear.

$$\text{scaleup} = \frac{\text{Elapsed time on P nodes indexing DB}}{\text{Elapsed time on P' nodes indexing DB'}}$$

[where P' > P and DB' > DB]

Fig 3-6. Scaleup metric

3.4.1.8 Load Imbalance (LI)

$$\text{LI} = \frac{\text{Maximum processing time on node}}{\text{Average processing time on node}}$$

Fig 3-7. Load imbalance (LI) metric

Hawking has defined a useful measure of Load Imbalance LI (Hawking, 1994) which we use in order to examine the load balance on our computations (see figure 3-7). An LI value of 2.0 is said to halve the effective speed of the parallel machine. The measure is a pessimistic one that starts from a figure of 1.0 and grows with the level of imbalance: the ideal load balance is near to 1.0. We may use this metric either on individual jobs in a task or several jobs: we have used it in both ways.

3.4.1.9 Space Costs

We record the size of the inverted file index in bytes (Mbytes or Gbytes). We also record the percentage of text for the index size. Any increase in certain aspects of an index is recorded such as increase in dictionary file size for the *DocId* partitioning method for increasing numbers of nodes (see chapter 6 on indexing). We also use the LI metric defined in figure 3-8 and apply it to file sizes instead of processing time.

3.4.1.10 Merge Costs

When creating the inverted file using an indexer it is not always possible to save the index directly to disk because of memory constraints. We use the metric defined in fig 3-8 to quantify the costs in relation to the whole indexing computation, averaging the time and taking the percentage. The ideal result for the metric is the lowest merge costs in percentage terms: the inference is that run time for the indexing computation will be lower.

$$\text{Merge Costs (\%)} = \frac{\text{Average time spent merging on P nodes}}{\text{Average processing time for indexing on P nodes}} * 100$$

Fig 3-8. Indexing merge costs metric

3.4.2 *Retrieval Effectiveness Metrics*

We use the standard measures of retrieval effectiveness, precision and recall, plus some filtering utility functions that were defined for TREC filtering experiments.

3.4.2.1 Precision

Precision is the quality of the documents presented to the user, i.e. how many are of the documents retrieved are relevant. It is calculated by taking the ratio of the number of relevant document retrieved over the total number of documents retrieved (Frakes, 1992). We use several extensions of this precision definition. One is average precision, that is precision averaged over each point in a rank where a relevant document is recorded: the more relevant documents higher up the rank the better the average precision. We also record precision at points 5, 10, 15 and 20 documents retrieved.

3.4.2.2 Recall

Recall is the measure of how many relevant documents are retrieved from the database. It is calculated by taking the ratio for the number of relevant document retrieved over a query over the total number of relevant documents for that query in the database (Frakes, 1992). We use this measure with routing and ad-hoc data but do not have enough relevance data to use the metric on the Web collections.

3.4.2.3 Filtering utility functions

The utility functions used are those applied in the TREC-4 (Lewis, 1996) and TREC-5 (Lewis, 1997a) conferences. The concept of utility is based on assigning a numeric value on each retrieved document, where relevant documents are given a positive value and non-relevant documents are given a negative value. The value of the utility can be adjusted according to the value users place on reading relevant or non-relevant documents. The basic method was to use three utility functions on each retrieved document, one requiring the filtering system to act in a high recall fashion (U3), one in a high precision fashion (U1) and one between these two (U2). The R variable in fig 3-9 is the number of relevant documents while N is the number of non-relevant documents. The goal is to achieve the highest utility possible for any utility function. The TREC-8 utility functions (Hull and Robertson, 2000) are defined in fig 3-10. We also use the utility scaling function defined in Fig 3-11 from Hull (1999).

```
U1=R-3*N
U2=R-N
U3=3*R-N
```

Fig 3-9. Filtering utility functions for TREC-4 and TREC-5

```
LF1=3*R-2*N
LF2=3*R-N
NF1=6*R^.5-N
NF2=6*R^.8-N
```

Fig 3-10. Filtering utility functions for TREC-8

$$u_s(S,T) = \frac{\max(u(S,T),\ U(s)) - U(S)}{\mathrm{MaxU}(T) - U(s)}$$

Where:

$u_s(S,T)$ = Scaled utitlity for systems S for topic T
$u(S,T)$ = Original utitlity for systems S for topic T
$U(s)$ = Utility of retrieving s non-relevant documents
$\mathrm{MaxU}(T)$ = Maximum possible utility score for topic T

Fig 3-11. TREC 7 utility Scaling Function

Chapter 4

PLIERS: A Parallel Information Retrieval System

4.1 INTRODUCTION

In this chapter we describe in detail the system we have built in order investigate the issues to be examined in this thesis. The system is called PLIERS (ParaLLeL Information rEtrieval Research System). The material for this chapter was originally published in MacFarlane et al (1999a), but is greatly expanded here. We give some background information on process topologies that is needed for understanding of the material in this chapter. We discuss important implementation issues for PLIERS such as the method of inter-processor communication and portability of the system. We then describe the algorithms used in each of the tasks under investigation in this thesis, namely indexing, search, passage retrieval, index maintenance and routing/filtering. We outline design decisions made in each of these tasks.

4.2 TOPOLOGIES

An important issue for any parallel program is the issue of process topologies, their configuration and how these topologies are physically mapped to processors in a parallel machine. The paradigm we use in all our programs is the distributed memory message passing one (Hwang, 1993: 551-4). We therefore split work amongst processes and communicate data between them where necessary. The topologies for each of the tasks we examine are explained in more detail in the discussion below, but we state here that our choice of process topologies are based on the belief that simple is beautiful. Parallel programs are complex entities that should not have any more complexity than is absolutely required. We generally use some form of a top or master process that communicates with the outside world, using leaf or slave processes to parallelize the task computations. This method of topology management is frequently used, both in and outside of IR: for example see chapter 2, particularly section 2.4.7 on the discussion of vector processing. We map leaf/slave processes to separate physical processors while placing master/top processes either on a separate physical node or on one where a leaf/slave resides if we have no more nodes to allocate.

4.3 IMPLEMENTATION ISSUES

4.3.1 *Programming with MPI - the Message Passing Interface*

The MPI standard (Dongarra et al, 1996) is the method we use for inter-processor communication in our programs. Various implementations of MPI have been used by PLIERS including CHIMP (Alasdair et al, 1994), MPICH (Gropp and Lusk, 1998) and ANU/Fujistu MPI (ANU, 1994). Semantics on some message passing commands can vary between implementations and this caused some slight problems. Various bugs in some of the implementations also caused some problems. Our experience with using different MPI implementations has generally been positive. We have found that the rank system is a useful abstraction particularly when used with collective operations (each processes is assigned a rank and messages are sent to or receive using this assigned rank): maintaining code is made easier than other methods such as Occam-2 (Bowler et al, 1989). Any topology change would require the re-write of hard wired collective operations. MPI is much more flexible. However this flexibility has its price. The requirement that implementation can vary part of the message passing semantics to cope with lack of buffering space led directly to a termination problem in the farming method for indexing described below. There is a fairly large set of routines and ideas to learn in order to use MPI to the full, much more so than Occam-2. We have not used PVM (Sunderam, 1990) so cannot compare it with MPI, but we would use MPI rather than Occam-2. PVM is an environment for connecting clusters of workstations together, and which provides services for message passing in much the same way as an implementation of MPI.

4.3.2 *Ease of Portability*

One of our secondary aims was to demonstrate the portability of our system. Using MPI (with GNU C) has allowed us to port our software to the different types of architectures described in chapter 3. We have demonstrated that our programs can run on two or more architectures which includes Sun SPARC, DEC Alpha and Intel Pentium based parallel computers. This process has not always been as simple and straightforward as we would have liked it to have been, and there have been some problems. Some of these problems relate not only to the architecture themselves, but also to issues on operating systems and MPI implementations (see section 4.3.1 above). An example of an operating system problem was that there is no virtual memory for the AP1000 and stack space is statically allocated. This restricts the use of techniques such as recursion which place extra demand on memory management: we had to move to iterative list processing in any case in MacFarlane et al (1999b).

4.4 INDEXING

Two types of index build methods are used: *Local* and *Distributed.* With *local build*, documents are kept on a *Local* disk and analysis is done on that *Local* disk only: this method is applicable to *DocId* partitioning only. The *distributed build* method works by distributing the documents to nodes from a single disk. It should be noted that replicated inverted files for routing/filtering were indexed on one node and distributed manually to local disks in the parallel machine.

4.4.1. *Indexing Topologies*

Our requirement for indexing topologies is to be able to support both partitioning methods under consideration as well as the two build strategies. The components of the topology must be reconfigurable in order to create different build types and numbers of inverted file partitions using different process combinations. Fig 4-1 shows examples of both types of builds using the *DocId* partitioning method, together with process to node mapping examples.

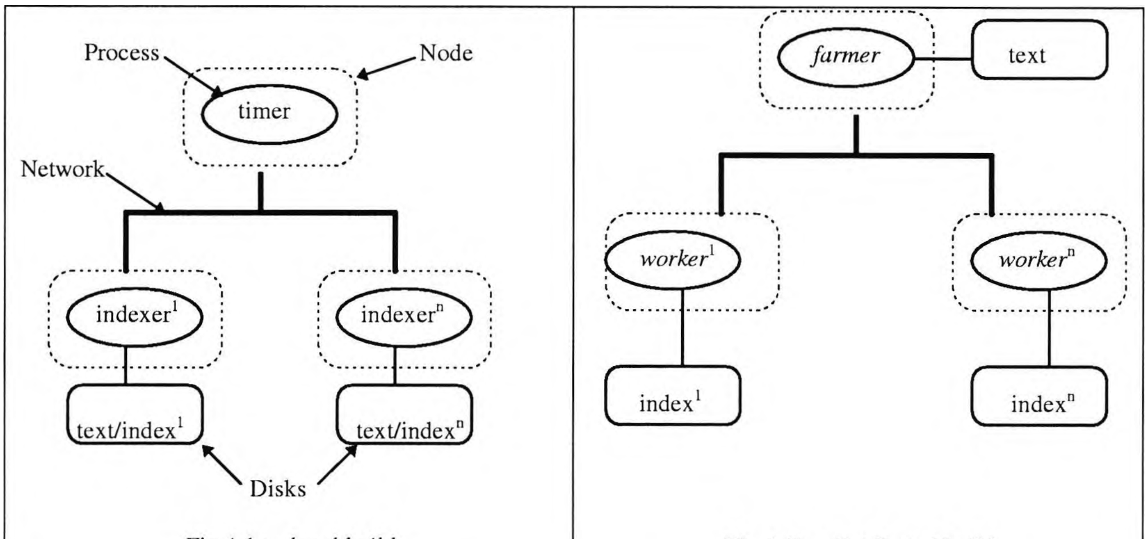


Fig 4-1a - *local build*

Fig 4-1b - *distributed build*

Fig 4-1. Build examples using the *DocId* partitioning method

The *local build* method for parallel indexing is a very simple topology requiring little communication (see fig 4-1a). Each indexer node runs independently with no need for communication between them (the function of the indexer is described below). This form of build is applicable to *DocId* only. The *distributed build* method uses the process farm paradigm (Bowler et al, 1989) and an example of the one proposed for indexing is shown in fig 4-1b. The structure in the example consists of a *farmer* and n *worker* processes whose function is described below. Fig 4-1 shows the contrast in the build methods particularly with regard to the distribution of text to be indexed. The difference between the two methods is that text is kept locally when the *local build* method is used, and kept centrally on a single disk when *distributed build* is used (see appendix A1 for an example of how this works). We use *local build* where a given collection could not be placed on a single disk (e.g. VLC2/WT100g).

Each method has its own advantages and disadvantages, and we leave the detailed discussion of such until later. The issue of communication is important here. It can be seen from both the diagrams and the descriptions above that some topologies will require a great deal more network resource than others. For example *distributed build* methods will require more communication than *local build* indexing in order to distribute text. Fig 4-2 shows an index topology example for *TermId* partitioning.

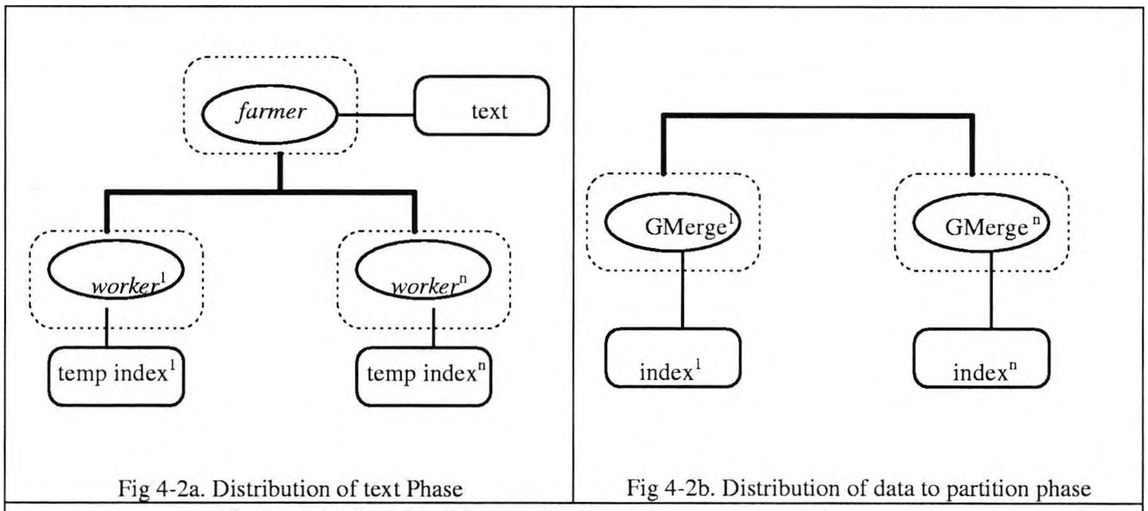


Fig 4-2a. Distribution of text Phase | Fig 4-2b. Distribution of data to partition phase

Fig 4-2. Distributed build example using the *TermId* partitioning method

4.4.2. *Distributed Build Topology Components*

In this section we describe the functions of the *farmer*, *worker* and *global merge* parallel processes. Note that there is only one *farmer* node, and a number of *worker* processes (which become *global merge* processes in *TermId*). Our reason for using this method is that it allows us to automatically distribute text to nodes: it has the disadvantage in that the method is more communication intensive than the *local build* method (see section 4.4.3).

4.4.2.1 Farmer Process

The *farmer*'s job is to distribute documents to the *workers* (see Fig 4-3). Essentially it distributes work as equally as possible to create the least amount of load imbalance possible. Single documents or files containing multiple documents can be distributed: the latter saves communication time. There is an initialisation stage where each *worker* is given its first initial document/file; after that *workers* are only given documents/files when they request them, i.e. send a message to the *farmer* asking for more work. When no more documents/files are left, a termination notice is sent to every *worker* process. Document identifiers are allocated individually if the granularity of parallelism is documents and in blocks if it is files. The document length cannot be recorded until the document has been analysed, and this data is sent to the *farmer* when a *worker* requests further work: this data is saved to disk when received. In an attempt to keep *workers* load balanced a request for work is serviced as soon as possible after it has been received so that *workers* who index small documents or files are not kept waiting for too long.

```
Distributed Initial set of documents/files to all workers

Loop no of files/documents
        get a request from worker i
        Case(request type)
            work request: send document/file to worker i
            id request    : send block of document id's to worker i
        EndCase
EndLoop
Loop until all workers have been terminated
        get a request from any worker i
        Case(request type)
            work request: send termination notice to worker i
            id request    : send block of document id's to worker i
        EndCase
EndLoop
```

Fig 4-3. Farmer algorithm for parallel indexing

4.4.2.2 Worker Process

The *worker*'s function is to break down the document into its constituent parts, i.e. terms, and perform some analysis on these terms, e.g. stemming using the methodology described below (see fig 4-4). If required, the position record is stored for each term using current values of accumulated data for field number, paragraph number, sentence number, word number and preceding stop words. After each word is found these values are updated. The *worker* creates and inserts this word/position data in a bucket: the method of storage for bucket elements is an AVL tree. In the case of *DocId* partitioning one bucket is used while 100 are currently used for *TermId*: words are hashed to a given bucket based on a dictionary (Cowie, 1989). The posting list is either created using the document identifier and the position record or updated by incrementing the number of positions and adding the position record to the position list. When any of the memory limits is reached, the results are saved on a temporary file on disk for each bucket. A *worker* then requests work from the *farmer* and waits for a new document/file to analyse. A termination notice is received when there are no more documents/file to be processed and the *worker* either saves the inverted file directly from memory if the inversion has fitted into memory, or merges the intermediate results to create the inverted file. Where *DocId* partitioning is required the process can stop here, if *TermId* is required then a *global merge* is invoked.

```
Loop until termination notice received
        Receive a document/file from the farmer
        Analyse document/file -> index
        If memory limits exceeded at any point during analysis
                then save index on disk
        Send request for work to farmer
EndLoop
If memory limits have not been exceeded
        Save index directly to create inverted file
Else
        Save current index to disk.
        Merge data saved on disk to create inverted file
EndIf
```

Fig 4-4. Worker algorithm for parallel indexing

4.4.2.3 Global Merge Process

This further process is only used for *TermId* partitioning (see fig 4-5). The *global merge* process has three phases; a heuristic is applied to choose the distribution of the files, the files are then transferred across the network to the required node and a second *Local* merge is initiated to create the final inverted file. The heuristic in the first phase works by calculating the average value for each of the 100 partitions and attempts to derive a distribution of buckets amongst nodes that is within a given criterion, currently with 10% of the average value: up to five iterations are used. The average chosen for distribution is to prevent a node being overloaded with data, while iterations were restricted to ensure the process of allocating terms to nodes was fast. The average value can be one of three variables on a bucket; word count (WC), collection distribution (CF) and term distribution (TF): we refer to these as term allocation strategies. When the distribution is generated it is used to transfer the files for that bucket to the node that has been allocated that bucket: this is done by gathering from all processes to the target process. The merge is then initiated on those transferred files.

```
Worker i

(Phase 1)
Exchange word frequency data with all other workers
Partition words amongst workers using required word distribution
        criteria (WC,TF,CF)

(Phase 2)
Loop no of partitions -> j
        If partition j belongs to worker i
                gather partition j data from all other workers
        Else
                Send partition j data to required worker
        EndIf
EndLoop

(Phase 3)
Merge data for Workers partition to create inverted file.
```

Fig 4-5. Global merge algorithm for parallel indexing (*TermId* only)

4.4.3. *Local Build Topology Components*

4.4.3.1 Timing Process

The only central process for *local build* is the timing process: it waits until all indexer processes are finished and saves the total elapsed time for the build. Our reasoning for using this method is to examine the scalability of our parallel data structures and algorithms: however because of its minimal communication it is the one most would choose in many circumstances.

4.4.3.2 Indexer Process

Each indexer process is a sequential index process that takes the function of the *farmer* and *worker* processes i.e. it reads in documents, breaks them down, adds them to the index creating intermediate indexes when a given set of criteria is met. The intermediate results are then merged to form one index for each node. The indexer process only communicates with the timing process when it has finished building the index: apart from that, its work is completely independent of any other process.

4.4.4. *Indexing Methodology*

For each index build we used a stop word list of 450 words supplied by (Fox, 1990) to filter out unwanted terms. All HTML/SGML tags are stripped from the text and ignored if not used for specific reasons such as identifying paragraphs <p> and the end of document </DOC>. Each identified word was put through a Lovins stemmer (Lovins, 1968), supplied by the University of Melbourne, and indexed in stem form. An exception to this was the use of a Porter stemmer on the Ziff-Davis collection. Numbers were not indexed. A large amount of in-core memory is pre-allocated in blocks by each indexing process, and documents are analysed until one of several criteria is reached: exhaustion of keyword block, posting block or position block space. When one of the criteria is satisfied, the current analysis is saved on disk as an intermediate index, so that the in-core memory can be used for the next set of documents. When all documents have been analysed, the intermediate indexes are merged together to create the final index and deleted.

4.5 PROBABILISTIC SEARCH

4.5.1. *Search Topologies*

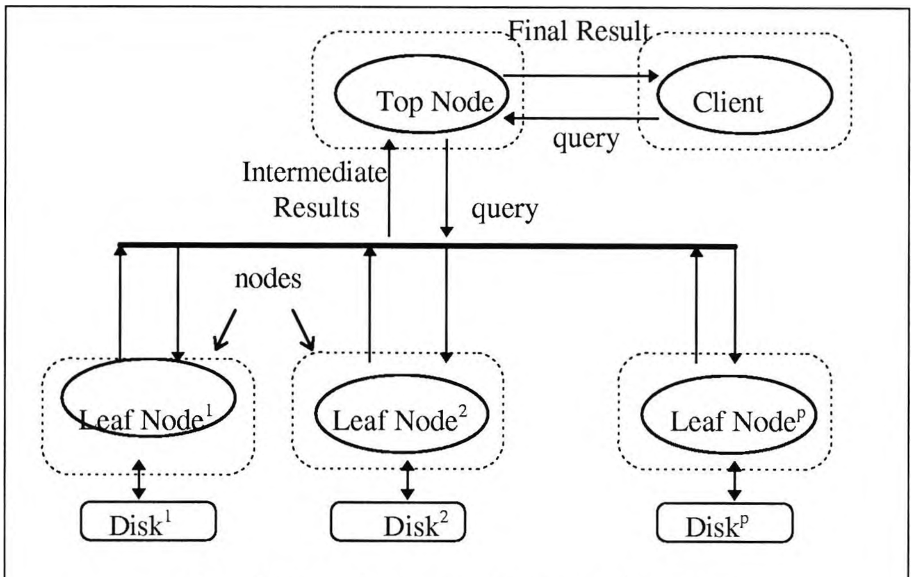


Fig 4-6. Example search topology configuration

In order to facilitate parallel searches on inverted files with differing data partitioning methods, we have implemented a generic search system that may be configured to use either partitioning method, consisting of a top node and one or more leaf nodes. The client process should be unaware of the distribution of data and retrieval responses with respect to effectiveness measures are identical on the different data partitioning methods. We also have a requirement that the mapping of logical to physical topologies should be as flexible as possible, but respect the distribution of data as generated by the indexer to ensure good performance. The topologies are designed to make efficient use of the shared nothing architecture. These components are described below, followed by a discussion on search topologies. An example of how the components are combined can be found in fig 4-6.

4.5.1.1 Top Node

The main task of the top node in a search topology is to act as the interface for a client to the topology. It accepts a query from the client, distributes it to all of its child nodes and awaits the results. Depending on the type of operation, it may sort the results ready for presentation to the client or merge partially ordered results from its child nodes.

4.5.1.2 Leaf Node

The leaf node looks after one partition of the inverted file. It keeps an in-core record of keywords that is searched when a query is received. The inverted lists are then built for each element of the query and merged together to form a final result set for this node. This result set (or sets) is sent to the top node. The number of leaf nodes is defined by the number of required inverted file partitions.

4.5.1.3 Discussion on search topologies

The example in fig 4-6 is a master/slave topology with a top node and n leaf nodes (each with its own disk). The query referred to in fig 4-6 is a set of keywords, while the sets are retrieved inverted lists. The service of a query is done as follows: the top node receives a query from a client and distributes it to leaf nodes 1 to n. The result set for that query is sent back to the top node, merging as necessary. In the example the top node is mapped to a separate node from the leaf nodes and the client is on another processor. We can map the top and client nodes to any of the available processors as we wish. For performance reasons we only allow one leaf node per processor. Mapping either the top node or client node either separately or together or with a leaf node also has performance implications. We describe the algorithm used and its impact on a parallel program with a given partitioning method in the next section.

4.5.2 *Probabilistic Search on Partitioned Inverted Files*

The term weighting model supported in PLIERS is the Robertson/Sparck Jones probabilistic model (Robertson and Sparck Jones, 1976) and the term weighting model used (BM25) is declared in chapter 3. There are four main tasks to search when using the probabilistic model for search. Firstly we retrieve the document sets from disk and place them in core. Once we have the document sets we can then assign a weight to each word/document pair. The sets can then be merged to create a single result set for the required keywords. This set is then sorted in descending order ready for presentation to the user. We describe both the implementation and implication of parallelism on each of these phases below.

The retrieval of document sets from disk is a straightforward process and the operation is identical irrespective of partitioning method used. The effect however can be very different: retrievals for *TermId* partitioning require only one I/O request per term, while *DocId* requires p requests (where p is the number of partitions), but the set transfer time per term will be shorter with *DocId* because of parallelism and short postings lists. It has been pointed out by Jeong and Omeicinski (1995) that there are performance trade-offs between I/O requests and data transfer time when retrieving sets from disk.

A very important aspect of term weighting is the issue of collection statistics with respect to partitioning methods. If we treat partitions of the inverted file as being part of a global database we need to exchange data between the partitions in order to ensure that the collection statistics are consistent for every weighting. For example in order to calculate a collection frequency weight we need to add all occurrences of a term from all partitions when using *DocId* partitioning: such addition is not necessary when using *TermId* partitioning since the term frequency is available in one partition. We therefore need to adjust our term weighting operations to suit the partitioning method being used: in the case of *DocId* partitioning an extra request to the nodes is needed in order to collate statistics for the search. It is possible to keep a global dictionary at the cost of extra space. Other statistics such as average document length and total number of documents in the collection are also affected. It should be noted that term weighting is possible on independent collections (Singhal, 1998; Cormack et al, 1998), but the discussion of this subject is outside the scope of our research: our aim is to address the issue of term statistics across partitioned inverted files and not the results merging problem.

Once weights for each element of the set have been generated we can then use a PLUS set weight operation that is a special case of union: if two document identifiers are in both sets we add their weights otherwise we insert the unchanged posting record in the result set. Again different types of operations are required in differing partitioned methods. Using *DocId* partitioning we can merge local results, do a sort and send only the top n documents from that leaf to the parent node. With *TermId* partitioning we cannot merge the sets for the result until the top node has received data from all leaf nodes. This would appear to give *DocId* partitioning methods an advantage over *TermId* in that less communication is needed.

The final phase is simple in *DocId*: a simple multiway merge will produce the top n documents required by the client. For *TermId* however we need to apply a sort to produce the top n documents. We can apply the sort either directly using a sequential sort or distribute the set elements amongst the leaf nodes and apply parallelism to it: we have implemented both methods. Given that *DocId* can apply a sort in parallel in the third phase without any communication requirement, the method accrues a further advantage over *TermId*. Only the top n documents identified by the weighting operation are presented to the user.

4.6 UPDATE AND INDEX MAINTENANCE

4.6.1. *Transaction Topologies*

Given that we want to service search and updates simultaneously, the transaction topology cannot differ too much from the search topology above in section 4.5. We therefore define top and leaf nodes that can handle both search operations and the update operations implemented. We describe the additional functionality needed by the nodes to support update operations above. Fig 4-7 shows an example transaction topology with the service of updates.

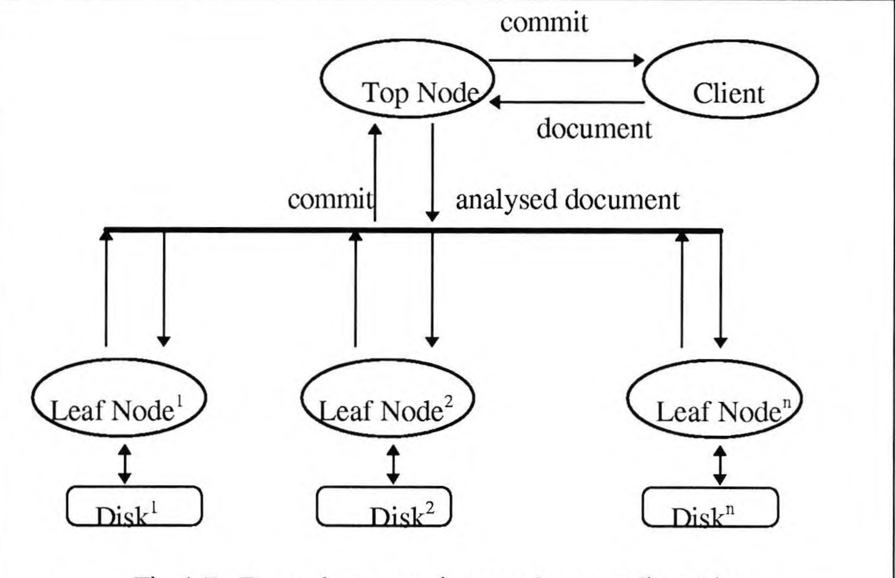


Fig 4-7. Example transaction topology configuration

4.6.1.1 Top Node

The top node being the interface to the topology, accepts new documents, breaks them down into their constituent words, and sends the index information to the relevant Inverted file fragment. An alternative would be to devolve analysis to the leaf node, at the cost of extra coding for little material benefit. The main issue here is that the top node must know what type of partitioning method is being used in order to send data to fragments accordingly. For example in *TermId* partitioning a bucket of words will be formed for each fragment of the Inverted file. These can be sent directly to the fragments. However with *DocId* partitioning, a decision must be made as to how new documents are allocated to the fragments. We assume that over a given period of time incoming documents that are distributed in a round robin fashion will give each fragment roughly the same amount of data, although it is unlikely to be evenly distributed. We therefore assign new documents to fragments using a round robin distribution method when *DocId* partitioning is used. For both types of partitioning method a confirmation of update completion must be received before a commit notice is sent to the client.

4.6.1.2 Leaf Node

The leaf node receives index data, and merges it with the fragment index data handled by that particular leaf node. This sequential process is identical to that described in the indexing section 4.4 above. Some collection statistics and document data must be shared amongst leaf nodes, e.g. collection size and document length. Each leaf has a *document map* structure described in chapter 3 which records such information. When a search transaction is received by the leaf, both the index and the in-core buffer are searched. If a reorganisation of the index is initiated new updates are added to a separate temporary buffer: this is searched as well if new queries are received by the leaf. Transactions are serviced while an index update is done, strictly interleaving reorganisation of terms and servicing queries/updates (see chapter 8 for more details). When the reorganisation is complete this temporary buffer becomes the main buffer.

4.7 PASSAGE RETRIEVAL

4.7.1 *The Sequential Algorithm*

The basic idea behind the implemented passage retrieval method is to iterate through contiguous sequences of text atoms (say a paragraph) and find the combination that yields the best weight for that document. This procedure is done on inverted files. The algorithm is shown in Fig 4-8.

```
Function do_passage( IN:  document data ) RETURN OUT: weight

 For( start=0; start < no_of_atoms; start++ )
  For( finish=start+INCREMENT_VALUE;
      finish < no_of_atoms && finish-start < MAX_PASSAGE_LEN;
      finish=finish+ INCREMENT_VALUE)
        If(at least one query term is in start and finish atoms)
          calculate weight for current passage
        EndIf
        if( current passage weight > largest passage weight )
          record details of current passage as best passage
        EndIf
  EndFor
 EndFor

 return best passage weight

END do_passage

(1st Phase)

        Obtain the top 1000 ranked documents for query terms

(2nd Phase)

        Get position lists for the terms in the query.
        loop 1000 documents
                call do_passage function to obtain best
                        passage for that document
        EndLoop
        Re-rank the top 1000 documents.
        Send top x documents to the client.
```

Fig 4-8. Algorithm for sequential passage retrieval

There are two stages to the sequential algorithm: retrieve the top ranked (say 1000 documents) and then apply the passage retrieval algorithm to all the elements of the top ranked set. Restricting processing to 1000 documents is a time saving process: in principle we could consider all documents. The first phase is a simple search as described in section 4.5 above. In the second phase a list of positions is obtained for each document and word pair in the query. We do not need to analyse full text since position data can be saved in our inverted file during the indexing process (see section 4.4). The passage retrieval algorithm is then applied, recording the best passage weight for each document. The processing requires the weighting of a document for terms in the query sets given the relevant position data, and iterating through the defined passage, recording the highest weighted passage. In the case of best match functions such as BM25 the requirements may also include the calculation of the passage length. This passage length is available from storage having being recorded at indexing time (see chapter 3). Once all documents have been processed the top set is re-ranked and can be presented to the user. We can reduce the time complexity for this method by setting a maximum passage length (the term MAX_PASSAGE_LEN in Fig 4-8). A figure of twenty atoms has been used

(Robertson et al, 1995) - see section 1.4.4 for the definition of atoms. The minimum number of atoms could be sensibly set to one (Robertson et al, 1995). A further refinement is to specify the number of increment steps for examined atoms: this can be altered by changing the INCREMENTAL_VALUE constant. It should be noted that if either the start atom or the finish atom contains no query term, we do not calculate the weight for that passage. In general there must be a shorter passage that would be better, given that the BM25 scoring function increases with reducing document length (see chapter 3, section 3.2.1).

4.7.2 *The Parallel Algorithm and Partitioning Methods*

In this section we describe the parallel implementation of the passage retrieval algorithm described above. There are a number of issues when considering the application of parallelism to the Okapi passage retrieval algorithm. In particular the issue of how the algorithm is applied to the inverted file with a chosen data partitioning method must be addressed. We define two methods of parallel passage processing to be considered: one that calculates passages on the basis of the database as a whole (distributed) and one where passages are processed locally (local). These methods are described below. We use a process topology described in more detail in section 4.5 above.

```
(1st phase)

        Obtain 1000 top ranked documents from the database.
        Broadcast top 1000 document id's to all nodes.

(2nd Phase)

        InParallel for P leaf nodes
           Retrieve position data for top ranked documents
                  for fragments query terms.
           Loop 1000 documents
                  If( document data is in the fragment )
                     call do_passage function to obtain best passage
                             for that document.
                  EndIf
           EndLoop
        EndParallel
        Retrieve and merge the document id weights from P leaf nodes
        re-rank the top 1000 documents
        Get the top x documents and send them to the client.
```

Fig 4-9. Parallel passage retrieval algorithm - version 1: distributed

4.7.2.1 Distributed Passage Processing

This method is a two-phase search which could be used for both types of inverted file partitioning (see fig 4-9). In the first phase we do a parallel search for the top 1000 documents as described in section 4.5. The identifiers of the top 1000 documents that are to be processed in the second phase are broadcast to all nodes. Passage retrieval is then applied to documents managed by a fragment and the result from the nodes is merged into one set. Note that the sequential algorithm for passage retrieval is reused in the parallel version. The partitioning method has a direct effect on the processing of passages. Passage retrieval works with *DocId* partitioning by letting each processor compute passage weights on each of their document sets independently. Our processing yields inter-passage parallelism and may be affected by load balance in different ways. For example documents may not be evenly distributed across nodes (some documents in the TREC collection (Robertson et al, 1995) can be large). Therefore processors with more documents or large documents may require much more processing than other less heavily loaded processors.

The way the method could work with *TermId* partitioning is to produce a set weight for each hit document, and accumulate the weights produced on different processors. We can consider this type of parallelism as intra-passage as the weights for any given passage is calculated across processors. A criticism of this method is that query size could cause poor load balance since some processors are likely to have more terms to process than others (see chapter 7 on probabilistic search results). The large computation needed for processing passages may well magnify the effect of the distribution. A further and more serious problem is the constraint on generating scores if and only if the start and finish atoms contain query words. Consider the scenario in Fig 4-10.

Atoms: A,B,C,D,E,F

Node 1: word: x is in A and D
Node 2: word: y is in C

Fig 4-10. *TermId* atom distribution for words

To generate the score for atoms A through F we need node 1 to generate component scores for weights for word x in atoms A and D, while node 2 does y in C. Unfortunately Node 2 does not have data for atoms A and D and will therefore not calculate its component of the overall score. We can consider two options to rectify this situation: 1) reach distributed agreement as whether to generate a weight for every passage: 2) relax the constraint and generate the whole search space. Both of these options have considerable problems. The distributed agreement method would require a considerable level of communication for every passage inspected, whether a score is generated or not. Much of this communication would be a waste of resources and puts an extra cost on the parallel algorithm over and above the requirements of the sequential version. Option 2 is hardly any better. Relaxing the constraint imposes an unacceptable increase in computation on the parallel algorithm. The storage space needed to keep the generated score for every passage inspected is high. The extra load at the top process to complete the passage processing is excessive. We do not therefore consider the passage processing method described in this chapter as being viable for use on indexes using the *TermId* partitioning method (we examine this more in chapter 5, section 5.4).

4.7.2.2 Local Passage Processing.

To overcome the potential problem with load balancing described above we can use an alternative method for passage processing (see fig 4-11). Each node does passage retrieval on a given number of documents say N: we can vary N according to our requirements. All passage processing is done locally on the node. As with the previous method, the *DocId* partitioning method only is used (and for the same reason). Once a node has received the query, no further communication is needed until the results have been produced. The effect is to reduce communication time at the expense of CPU time depending on documents processed (or how big or small N is). Given that the passage retrieval can be applied to more documents, the method also gives us the potential to examine more of the search space. With each additional processor we can examine more or less of the search space as required. In this method a total of N*P documents is examined for possible good passages (where P is the number of leaf nodes), although they are not in general the top ranked N*P documents in the collection.

```
InParallel for P leaf nodes
        (1st Phase)
        Obtain N top ranked documents for the fragments
                query terms.

        (2nd Phase)
        Get position lists for the terms in the query.
        Loop N documents
                call do_passage function to obtain best
                passage for that document
        EndLoop
        Re-rank the top N documents.
EndParallel
Retrieve top x documents from P leaf nodes.
Send top x documents to the client process.
```

Fig 4-11. Parallel passage retrieval algorithm - version 2 local

4.8 TERM SELECTION FOR ROUTING/FILTERING

4.8.1 *Approaches Taken by OKAPI at TREC*

Previous research has been done through Okapi within the framework of the TREC conference. The approach taken in these experiments is to apply hillclimbing techniques (Tuson, 1998) of various kinds in order to optimise queries for routing/filtering. We describe the approaches taken so far below in TRECs 3,4,5 and 6 (Robertson et al, 1995; Robertson et al, 1996; Beaulieu et al, 1997; Walker et al, 1998) in order to give some background on the research.

Okapi at TREC-3 (Robertson et al, 1995) used a single-pass algorithm for term selection. The algorithm started with the top three terms when building the *term set*. The terms were considered one at a time, in sequence given by an ordering criterion for these terms. The algorithm did not use backtracking: that is that once a term has been accepted/rejected it was not considered again. Each subsequent term in the top τ terms was added to the query and evaluated against the training set; the term was retained in the *term set* if and only if a given acceptance criterion was reached. The acceptance criteria used were increases in average precision, r-precision, or recall or a combination of these.

Okapi at TREC-4 (Robertson et al, 1996) introduced the concept of adding only, removing only and add/remove terms iteratively from the query when selecting terms on three new algorithms. The *add only* method is similar to that used in TREC-3 described above. In the *remove only* method the initial query contains all terms and reformulation is done by taking terms off one by one and evaluating them against the database. The *add/remove* method is a combination of the latter, allowing backtracking. These strategies were applied to three algorithms; Find Best (FB), Choose First Positive (CFP) and Choose All Positive (CAP). With the FB algorithm each term was examined in one iteration and the term yielding the best score either added/removed from the query. CFP works by adding/removing the term if it is the first term that increases the score by an amount greater than a predetermined threshold. CAP is an extension of FB/CFP and works by including/excluding all terms that increase the score beyond a given threshold. The Find Best and Choose All Positive algorithms are Steepest-ascent hillclimbers while Choose Some Positive is a First-ascent hillclimber (Tuson, 1998). It was found that the best acceptance criterion was TREC average precision, but a number of other criteria have been tried.

Okapi at TREC-5 (Beaulieu et al, 1997) introduced the notion of varying document weights during the selection process. The CAP algorithm from TREC-4 was used and extended. Weights were reduced by a factor of 0.67 or increased by a factor of 1.5. A number of different strategies were applied for term weight variation, e.g. apply weight variation on a single non-included term at the original weight, lower weight then the higher weight, choosing the weight that gave the best score increase. More recently the technique of simulated annealing has been applied to re-weighting at OKAPI at TREC-6 (Walker et al, 1998) with disappointing results.

Three databases may be used for the Okapi routing/filtering method: an extraction database, a selection database and a comparison database. However extraction and selection functions can be done on one database. The extraction database was used to extract terms in the first level selection process (i.e. to create an initial set of terms); the selection database being used for the term optimisation process; the comparison database was used mainly for evaluating the various algorithms. This represents the first phase of Okapi at TREC experimentation: a further phase was used with Okapi splitting the database into extract and select for the final submissions to TREC (we did not do this phase of term selection).

4.8.2 *Description of the Sequential Algorithms*

It is useful to think of the term selection algorithms described above on two different levels: a concrete level and an abstract level. The concrete level describes what decision is made to select a given term by applying an evaluation procedure and getting a score (see sub-section 4.8.2.1). We can then use these scores to choose a best term at the abstract level (see sub-section 4.8.2.2).

4.8.2.1 Evaluations

In order to facilitate the discussion of the sequential algorithms in the next section we describe in detail what term evaluations are and the data they process. We define a *term set* as a set of keywords that has been chosen from a set of relevant documents during relevance feedback. We define a *document identifier* set as a set of documents in which a given keyword in the *term set* occurs. Each term in the *term set* has its own *document identifier* set and each element of the *document identifier* set has a pre-computed weight. A pairwise set operation can either be a merge or subtraction. For term addition we use a merge operation on the *document identifier* set while for term deletion we use a subtract operation. The merge operation accumulates weights for a given document, while the subtract operation reduces the total for a document.

In order to do evaluations we need a list of relevant documents. This list takes the form of a *document identifier* set. The first phase of an evaluation is to apply a set operation to the *document identifier* set of the current term and an accumulated set, forming an intermediate set. The accumulated set is the merge of all *document identifier* sets related to terms chosen so far in the process. Using the relevance judgements we mark each *document identifier* in the intermediate as being relevant or non-relevant. This intermediate set is sorted in decreasing order of document weight and the required evaluation criterion is used to produce a score. This score is compared with the score for the accumulated set and the current term is either put in or removed from the query if and only if the score is increased by a required amount or other criteria. If the term is retained/removed the intermediate set becomes the accumulated set.

4.8.2.2 Algorithms

The first part of the sequential algorithm extracts a set of terms from a set of relevant documents and chooses a top set of terms using the Term Selection Value (Robertson, 1990): this is referred to as the term pool. The term pool generation is done before the optimisation process. An initial base set of terms is formed using the top three terms in the term pool: a set merge creates an accumulated *document identifier* set from the data of base set terms. One of the algorithms described above is applied iteratively to an evaluation set of terms using one of the different operations also described above in section 4.8.1 (e.g. FB with add/remove and re-weighting) to select terms for the query. The evaluation set is the term pool minus the base set. We apply the algorithm until some stopping criteria is reached (see Figs 4-12 and 4-13).

```
Load term pool
Generate base and evaluation term sets
Obtain relevance information for a topic
Form accumulated set from base set
Loop while no stopping criteria is reached
    Loop list of terms in evaluation set
        Evaluate an operation on the current term
        If operation is successful
            Record current term as best term.
            If algorithm is CFP
                Leave inner loop
            EndIf
        EndIf
    EndLoop
    Update query with best term
    Check the stopping criteria
Endloop
```

Fig 4-12. Sequential FB/CFP algorithm

```
Load term pool
Generate base and evaluation term sets
Obtain relevance information for a topic
Form accumulated set from base set
Loop while no stopping criteria is reached
    Loop list of terms in evaluation set
        Evaluate an operation on the current term
        If operation is successful
            Add current term to query
        EndIf
    EndLoop
    Check the stopping criteria
Endloop
```

Fig 4-13. Sequential CAP algorithm

Thus there are two iterations involved in term selection: an outer iteration after which a term or *term set* is selected and an inner iteration in which the evaluations are done. A further evaluation can be done on a different set of documents, to check the validity of the term selection.

4.8.3 *Description of the Parallel Algorithms*

4.8.3.1 Set Parallelism

An approach to applying parallel computation to term selection is to think in terms of *term sets* and what needs to be done to the Okapi algorithms to reduce their run time. Since the evaluation of operations on terms can be done independently we can distribute the evaluation set to a number of processes. These processes can then apply the required Okapi like algorithm in each inner iteration to each sub-set of the evaluation set. Thus by applying inter-set parallelism to the evaluation of terms in the evaluation set, we aim to speed up each inner iteration. We use a method of parallelism known as the *domain decomposition strategy* (Glover and Laguna, 1997: 260): the search space is divided amongst available processors. One of the advantages of this method is that communication costs are kept to a minimum as processes involved in evaluating terms do not need to communicate to complete their task: however there is an overhead associated with checking the stopping criteria in every outer iteration. This overhead involves both the retrieval of the best yielding term from all slaves and broadcast of the best term data included back to the slaves. We could consider the use of intra-set parallelism, that is applying parallelism to an individual evaluation. This method however would increase communication costs dramatically and it is not clear that the benefit gained by parallelism would offset this extra cost (this issue is addressed in chapter 5, section 5.5). We therefore concentrate on inter-set parallelism methods.

4.8.3.2 Parallelization of the algorithms

The combinations of algorithms and operations currently implemented are: Find Best, Choose First Positive and Choose All Positive algorithms with *add only*, *remove only*, *add/remove* operations and re-weighting (12 combinations in all). See section 4.8.1 (third paragraph) for a description of these algorithms and operations. It should be noted that the CAP algorithm is a purely sequential algorithm to which inter-set parallelism cannot be directly applied as the results are the cumulative effect of evaluations in one iteration. However the CAP algorithm can be applied to each sub-set of the evaluation set and we refer to the revised version as the Choose Some Positive (CSP) algorithm: we can regard CSP as a compromise between the FB/CFP algorithms and CAP algorithm. In the CSP algorithm the best yielding sub-set of the evaluation set from one process only is chosen. CSP is implemented in terms of CAP. Choose First Positive differs slightly in that it is possible that a better term could be chosen in one inner iteration for each smaller sub-set of the evaluation set. It is possible the terms selected by the Find Best algorithm may differ slightly over runs with varying numbers of processes, possibly affecting the evaluation score. This is because two or more terms may have the same effect when applied to the query and the term that is chosen first amongst these equal terms will be the term used: the term returned by the fastest process will therefore be chosen. When the number of processors equals the number of evaluation terms, all term selection algorithms are identical; i.e. they all reduce to Find Best.

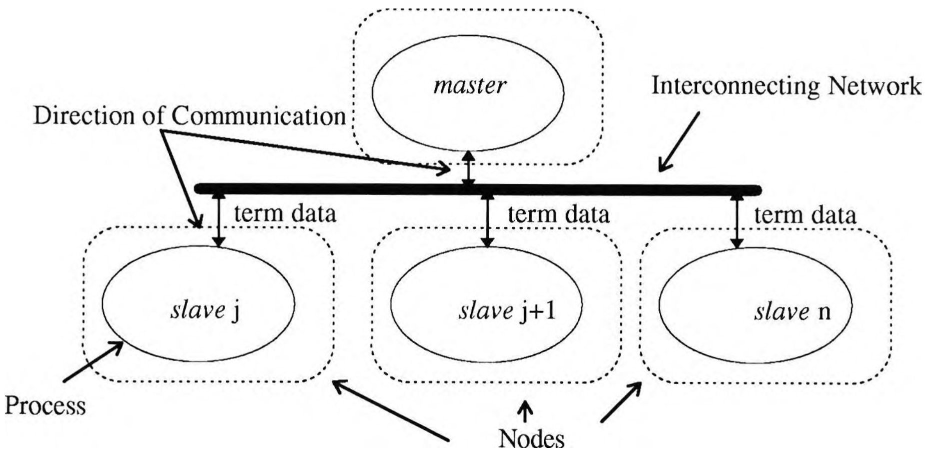


Fig 4-14. *Master/Slave* router topology

4.8.3.3 Types of Processes

There are two types of processes involved in parallel term selection: the *master* router and a number of *slave* routers. This topology was chosen for its simplicity. The *master* router creates the base and evaluation sets and handles the results from slaves after each inner iteration, checking the stopping criteria. The *slave* router applies the chosen selection algorithm (with the required operation) to its sub-set of the evaluation set, and communicates this information to the *master* router (see fig 4-14 together with figs 4-15 and 4-16).

4.8.3.4 The Parallel Algorithms

The parallel algorithms are shown in figs 4-15 and 4-16. The symbol $\rightarrow$ signifies the direction of a communication between process(s). Synchronisation between all processes is done in the stopping criterion at the end of each inner iteration. At this synchronisation point data is exchanged between the *master* and *slave* router processes in order to select the best term(s) from one of the processes. All *slave* processes then have the same term set data on which to select terms in the next inner iteration. This synchronisation point is a potential performance bottleneck both in the communication needed to transfer data and having a sequential section that cannot be parallelized.

```
Load term pool for topic
Generate base and evaluation term sets
Obtain relevance information for a topic
Form accumulated set from base set
Broadcast accumulated set, evaluation set and
        relevance data to N slaves [master→slaves]
Loop while no stopping criterion is reached
  InParallel ForAll j =1 to N Slaves
    Loop list of terms in jth evaluation sub-set
      Evaluate an operation on the current term
      If operation is successful
        Record current term as best term
        If algorithm is CFP
          Leave inner loop
        EndIf
      EndIf
    EndLoop
  EndParallel
  Get best term from N slaves [slaves→master]
  Check the stopping criterion
  If stopping criterion is not reached
    Update query with best term
    Broadcast new best term data to N slaves
                                    [master→slaves]
  InParallel ForAll N Slaves
    Update accumulated set with best term data
  EndParallel
  EndIf
Endloop
```

Fig 4-15. Parallel FB/CFP algorithm

```
Load term pool for topic
Generate base and evaluation term sets
Obtain relevance information for a topic
Form accumulated set from base set
Broadcast accumulated set, evaluation set and
        relevance data to N slaves [master→slaves]
Loop while no stopping criterion is reached
  InParallel ForAll j =1 to N Slaves
    Loop list of terms in jth evaluation sub-set
      Evaluate an operation on the current term
      If operation is successful
        Update query with current term.
      EndIf
    EndLoop
  EndParallel
  Get best best term from N slaves [slaves→master]
  Check stopping criterion
  If stopping criterion is not reached
    Broadcast new best term data to N slaves
                                    [master→slaves]
  InParallel ForAll N Slaves
    Update accumulated set with best term data
  EndParallel
  EndIf
Endloop
```

Fig 4-16. Parallel CSP algorithm

4.8.3.5 Data Placement Strategies

Two strategies for the distribution of inverted files have been implemented. In strategy 1 the *master* only has access to inversion (on-the-fly distribution) while in strategy 2 inverted files are replicated across all processes, *master* and *slave* (fig 4-17). The parallel architecture used for the strategy 1 distribution method was the distributed memory architecture used in many applications for parallelism. Strategy 2 used a shared nothing architecture that is described in chapter 3. We used two different distribution methods for the evaluation set. For strategy 1 we distributed evaluation set terms on a round robin basis using the Term Selection Value criterion. For strategy 2 we sorted the query in increasing collection frequency per term, and then used a round robin distribution method for the evaluation set. It should be noted that once terms in the evaluation set were given to a node, the term stayed on that node: we did not use any dynamic re-distribution of terms.

We give a brief discussion of why partitioning methods under consideration in this thesis are unsuitable for use in applying parallelism to the term selection algorithms discussed (we give a formal treatment in the synthetic models chapter). The *DocId* method for inter-set parallelism would not be feasible due to the amount of communication needed: we would need (N-1)*T messages in order to exchange data for all the terms in the query in order to optimise it (N is the number of nodes, T is the number of terms in the query). This problem occurs because unique term information is spread across all fragments. Such a problem is not one that *TermId* suffers from: evaluations on a given term can be done without the need for any communication on its related data. However the problem with *TermId* is that it has reduced flexibility in distributing terms in the evaluation set compared with the strategies described above. As term data is fixed on one node, there is no guarantee that we would get good load balance as evaluation set terms may not be distributed equally amongst nodes. We discuss the suitability of these data placement schemes in chapter 5, section 5.5.

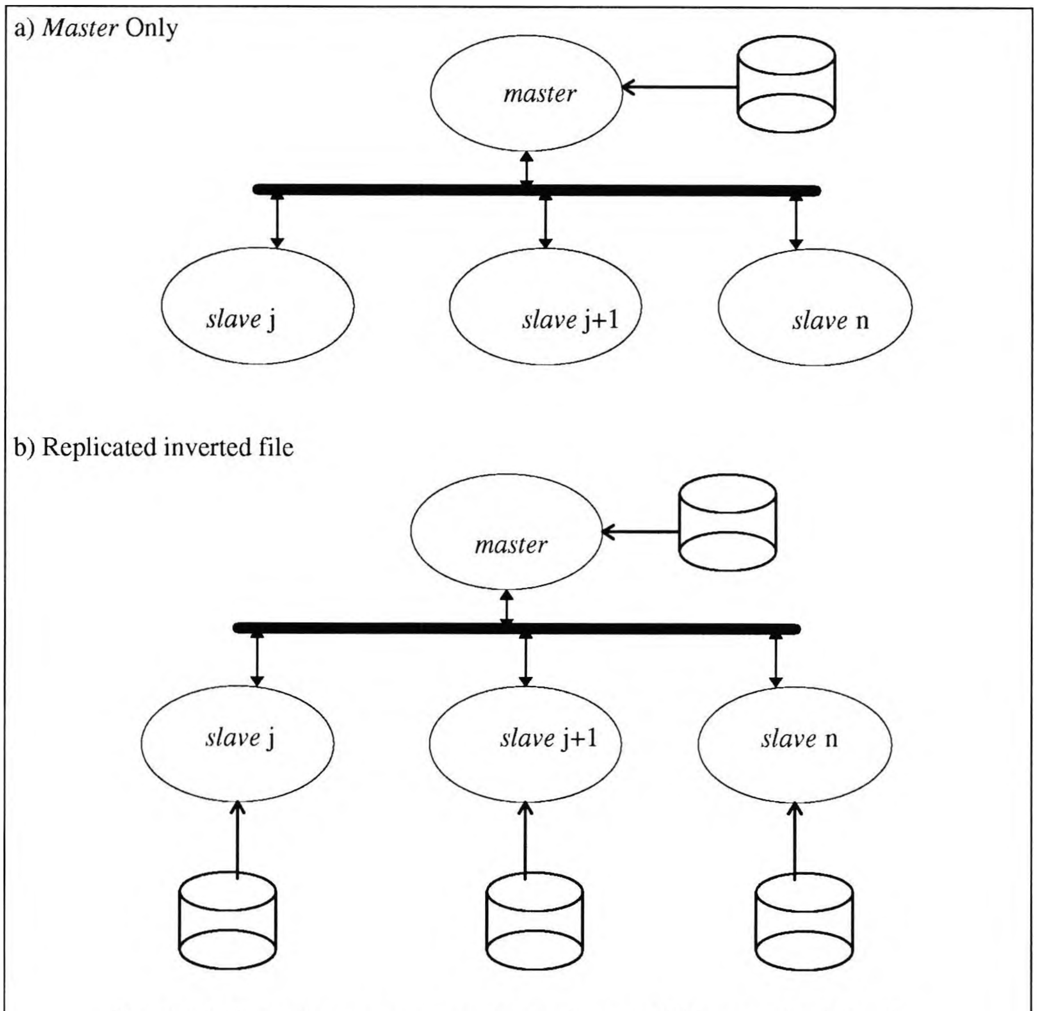


Fig 4-17. Data placement strategies for inverted files: routing/filtering

4.9 SUMMARY

In this chapter we have described the implementation of our parallel IR programs and have outlined important design decisions for each of the tasks being examined in this thesis. We stress the importance of simplicity in process topologies to reduce complexity and we have applied this principle in all of the implemented tasks. We have built three parallel programs in order to complete the research for this thesis: one for indexing, one for search/update/passage retrieval and one for routing. We have also built sequential versions for indexing and search programs for comparison. Client/Server programs for update and routing experiments are easily configured from parallel programs.

Chapter 5

Synthetic Models For Performance on Distributed Inverted Files

5.1 INTRODUCTION

Many previous performance models for IR only cover some aspects of the ideas and concepts discussed in this thesis. Those models which provide a general performance overview of IR do not deal with the problem of distribution (Cardenas, 1975; Fedorowicz, 1987; Wolfram, 1992a and 1992b). Much of the work described in the literature on the subject which does look at distribution either tackles one task (Jeong and Omiecinski, 1995; Tomasic and Garcia-Molina, 1993a and 1993b) or one aspect of a task such as the consideration of only one distribution method (Ribeirio-Neto et al, 1999; Hawking, 1996). In this chapter we provide synthetic models of performance in order to compare distribution methods for all tasks under consideration in the thesis. We also wish to fill some of the gaps in the literature. However, due to practicalities we do not study absolute performance of the tasks, and our emphasis is restricted to the derivation of synthetic models that can only be used for comparative purposes. Not having to address the issue of absolute performance simplifies the process of modelling greatly. While our primary aim in this chapter is to produce models which are strong enough to compare distribution methods and make choices between them, we also try to look beyond this simple requirement in the thesis. We would like models which are good enough to predict the relative difference between data distribution schemes. We would also like to be able to make generic statements about parallel IR performance beyond the algorithms and architectures which we examine in this thesis: this may be difficult to do given the range of systems described in the literature (see chapter 2). These issues will be addressed throughout the rest of the thesis by examing the empirical results gathered. In chapter 4 we asserted that some distribution methods for the passage retrieval and routing/filtering tasks were not viable: we address the issues formally in this chapter.

The format of the models is functional. This allows us to specify equations and reuse them in other defined equations. This makes it easier to replace various aspects of a given model in order to study different type of methods not under consideration in this thesis such as query processing optimisation and compression. We attempt to make our models as generic as possible. All functions return a single figure in abstract time. The functions only take variables as arguments: we do not specify higher order functions. We have a number of general variables for the models which are declared in table 5-1.

T_{cpu}	:	CPU time (for some operation).
$T_{i/o}[x]$	:	I/O time [Components: $1T_{seek}$ + x $T_{trans;}$ T_{seek} : Time to seek for I/O T_{trans} : Time to transfer data for I/O]
T_{comm}	:	Communication time
T_t	:	Abstract Time
P	:	No of nodes in a parallel machine
LI[P]	:	Load imbalance estimate at P processors

Table 5-1. General variables for the synthetic models

We make a number of assumptions in the general variables which impact (with varying degrees) on the synthetic models. We assume a low latency network in order to simplify the modelling of communication (otherwise we would have to break down T_{comm} using a $T_{comm}[x]$ format). For I/O we do allow two forms as blocks of data can be either static or dynamic. The Ti/o form of the variable can be used if fixed size blocks are transferred, and it is safe to assume that the balance between transfer and seek time is constant. We use the Ti/o[x] where variable sized blocks are transferred and the balance between seek and transfer time must be an integral part of the modelling process. For seek time we assume that an I/O request entails a single disk head movement. We assume an accumulated increase in load imbalance (variable LI[P]), at a rate of 0.015 for all synthetic models (values used are listed in appendix A5: table A5-1). It is difficult to know what the load balance will be for the parallel version of a particular task, without running a program and measuring the imbalance. We take this approach to provide a reasonable level of load imbalance for a given parallel machine size. For a given model we assume that the same parallel machine is used, that is the communication, CPU and I/O costs are identical across nodes in the machine: we do not address the issue of heterogeneous parallelism in the models.

The algorithms and methods modelled in this chapter are those described in chapter 4. Lookup values declared in the form x[y] (e.g. LI[P]) are not recorded as parameters but global variables. The scope rules for any declared variable are the normal ones found in most programming languages: variables declared locally take precedence over global ones. Sequential and parallel models are declared for all tasks. Simplifying assumptions for each of the models is declared in the relevant sections. The format for discussing models in each task is as follows; variables and constants for the model are declared, a brief description of the model is given, and the results using that model are examined. Details of how the models were constructed and derived can be found in appendix A5.

5.2. MODELS FOR INDEXING

Table 5-2 declares variables and constants for indexing models. We assume in this model that there is a strong link between word data and the underlying data structures (for postings and/or positions), such that we are able to abstract these data structures away and define models just using document and word information. This allows us to simplify the modelling considerably.

d	:	Number of documents in collection.
n	:	Average number of words in a document.
BSIZE	:	Blocking Size (number of words which can be stored in-core during indexing).
f	:	filesize in text words (no of documents in a file times
average		
		size n words)

Table 5-2. Variables and constants for indexing models

5.2.1 *Description of models for indexing*

We identify three distinct parts of time for indexing (see chapter 4, section 4.4): a) analyse the documents, b) save intermediate results to disk and c) merge intermediate results to create Inverted file. The sequential cost model $INDEX_{seq}$ for this process is declared and constructed in appendix A5-1. For *local build* the construction of the model ($INDEX_{Local\_Docid}$) is done by dividing $INDEX_{seq}$ by the number of processors in the theoretical parallel system, multiplied by load imbalance (LI[P]). We ignore any communication in this model as it is not significant. For the *distributed build* models of parallel indexing ($INDEX_{Distr\_Docid}$ and $INDEX_{Distr\_TermId}$) there are two extra costs associated with the methods (see chapter 4, section 4.4.2). With both partitioning methods there is the cost of distributing text data to the worker processes. With *TermId* partitioning there is also a further global merge phase which entails the transmission of data together with a further merge of this data. The parallel models are declared and constructed in appendix A5-2.

5.2.2 *Comparative results using indexing models*

The values of instantiated variables are declared in table 5-3. These variables are derived from the BASE1 collection, while the time variables are chosen to reflect the approximate balance between those variables in time T_t. Communication time assumes a fast network. We take these values and produce two sets of results: one with the collection kept static in order to compare the parallel versions of indexing (see fig 5-1) and one with the number of processors held constant in order to compare scaling between sequential and parallel versions (see fig 5-2).

d	:	187,000
n	:	460
BSIZE	:	716,000
f	:	171,697
T_{cpu}	:	0.01
$T_{i/o}$	:	0.015 (block sizes are constant, ignore x)
T_{comm}	:	1

Table 5-3. Values used for indexing models

From fig 5-1 it can be seen that there is an advantage in theory in using *DocId* partitioning over the *TermId* method for indexing in that the models for the former predict better performance over the latter on all machine sizes. It should be noted that the model predicts a narrowing of the gap between partitioning methods with increasing machine size (we will examine this further in the conclusion). There is little difference between local and distributed builds in *DocId*, but we have assumed a high bandwidth network in the instantiated models (the graph for local build *DocId* is obstructed by distributed build *DocId* as there is virtually no different in theoretical time between them). If we assumed a much lower bandwidth network, there would be a clear difference between the builds and *TermId* would not compare well with either of the *DocId* methods.

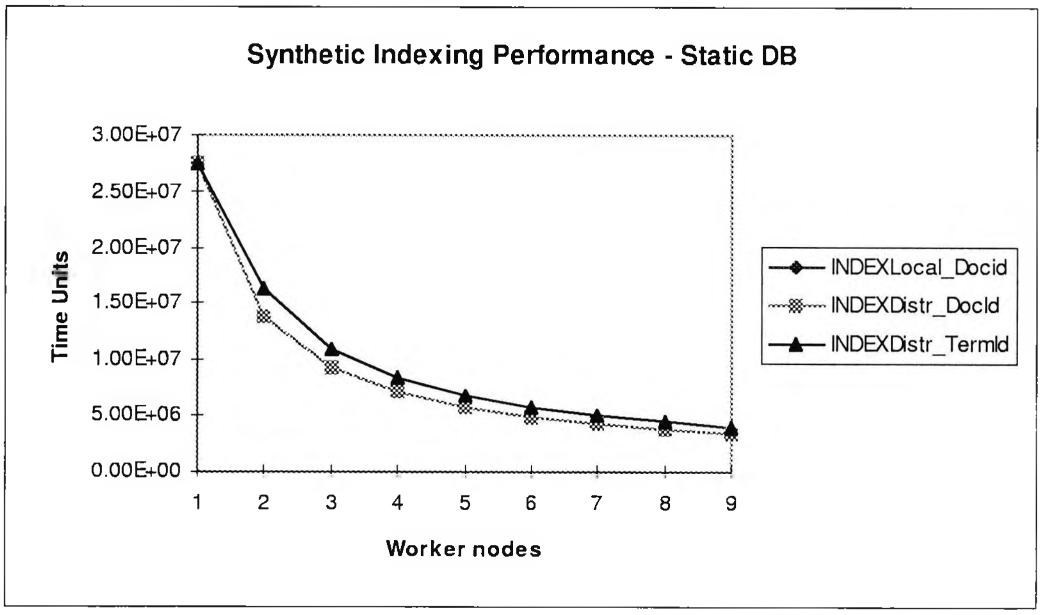


Fig 5-1. Synthetic indexing performance on 1 to 9 worker nodes

Fig 5-2 shows results for scaling on the synthetic indexing models. The number of worker nodes used in parallel models is 10. The comparison predicts that in theory, parallel runs would outperform sequential runs and that the gap would increase as the number of documents in the collection is increased. The comparison between the two partitioning methods also predicts that *DocId* using any build has a clear theoretical advantage over the *TermId* on large collections. Note that the assumption on high bandwidth predicts that there would be little difference between *local* and *distributed* builds when *DocId* partitioning was used.

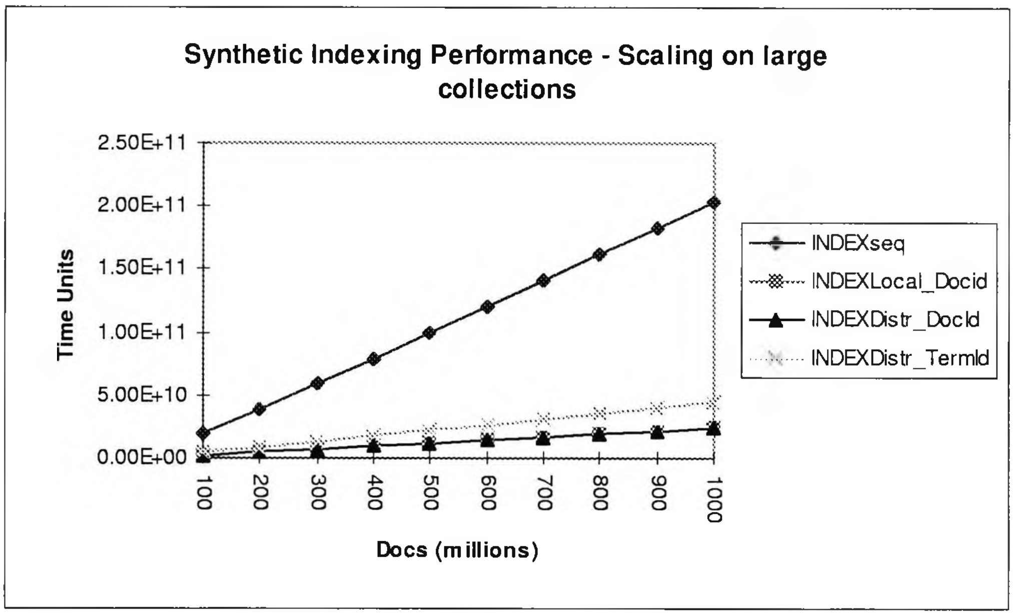


Fig 5-2. Comparison between indexing models on larger databases (scaling)

5.3 MODELS FOR PROBABILISTIC SEARCH

s	: Average set size
q	: Number of terms in a query
R[q,s]	: Final result set size lookup table determined by set Increase
SSIZE	: Size of set transferable in one comms invocation
P[q]	: Usable processors on query size for *TermId*: (P[q] <= P)

Table 5-4. Variables and constants for probabilistic search models

Table 5-4 declares the variables and constants for the synthetic performance models of probabilistic search (we model the search methods discussed in chapter 4, section 4.5). It should be noted that unlike the indexing models, data transferred from disk can be of varying size and we must therefore must break down $T_{i/o}$ into its components parts. In the lookup table R[q,s], we assume that the result set size will asymptotically approach the maximum set size for a collection with increasing the number of terms in a given query. When a given number of terms in a query is reached, we assume that the result set cannot be increased beyond this maximum set size. We introduce another form of counting processors in a system for *TermId* search, in the form of P[q]. This is because the number of terms in the query may be less than the node count of the parallel machine (e.g. the user may enter a single term query on an 8 processor system). The inequality P[q] <= P shows this aspect of *TermId* search formally. It is possible that all the terms in a query could be distributed to a single node: we therefore assume that terms are equally distributed to processors, i.e. the term allocation mechanism is ideal (see chapter 4, sub section 4.4.2.3). Similarly we assume that the distribution of documents when using *DocId* search is ideal.

5.3.1 *Description of search models*

In section 4.5.2 (chapter 4) we identified four discrete computations for probabilistic search, namely a) loading the keyword sets, b) weighting these sets, c) merging the sets into a single set and d) sorting this set to obtain the final result for presentation to the user. A function is declared for each these computations and the whole forms the model SEARCH_{seq} which is constructed and declared in appendix A5.3. The form of the parallel models is the same as the sequential model, allowing us to re-use and substitute functions in those models as required (see appendix A5.4). There is however a further communication cost for the parallel models. We assume in the *DocId* partitioning model (SEARCH_{docid}) that latency is main communication problem since little data is transferred: under these conditions, we can take the expected communication time, T_{comm}, to be the same for every message. We define two models for *TermId*, one which models a sequential sort ($\text{SEARCH}_{termid1}$) and one which models a parallel sort ($\text{SEARCH}_{termid2}$). The $\text{SEARCH}_{termid1}$ model requires more communication than SEARCH_{docid} and we assume here that data transfer is more of a problem. The $\text{SEARCH}_{termid2}$ model must model the scatter/gather of the data for the parallel sort. In both *TermId* models we substitute the alternative counting method for processors P[q] for P in the function invocations. The I/O is modelled differently in *TermId* models due to the effect of the seek/transfer balance (see chapter 4, section 4.5.2).

5.3.2 *Comparative results using search models*

SSIZE	:	4000
q	:	2.5
s	:	7791
T_{cpu}	:	0.003
T_{trans}	:	0.015
T_{seek}	:	0.1
T_{comm}	:	1

Table 5-5. Values for search models

Table 5-5 shows values used for the synthetic search models. The value for the s variable is an estimate based on data from the BASE1 collection. The time values are estimates which reflect the approximate balance between the different aspects. The value for q is used because users tend to submit just over two terms per query (from our experiments using real web data to be described in chapter 7, section 7.7.2). The SSIZE variable is chosen to represent the approximate costs during communication of one communications invocation. The results of applying these values shown in fig 5-3 demonstrate that in theory the *DocId* partitioning method would perform better than the *TermId* method using either a parallel sort ($SEARCH_{termid2}$) or sequential sort ($SEARCH_{termid1}$). The comparative results also predict that the *TermId* method with parallel sort will outperform the algorithm with a sequential sort by a substantial amount: the synthetic model predicts that a sequential sort will be a bottleneck.

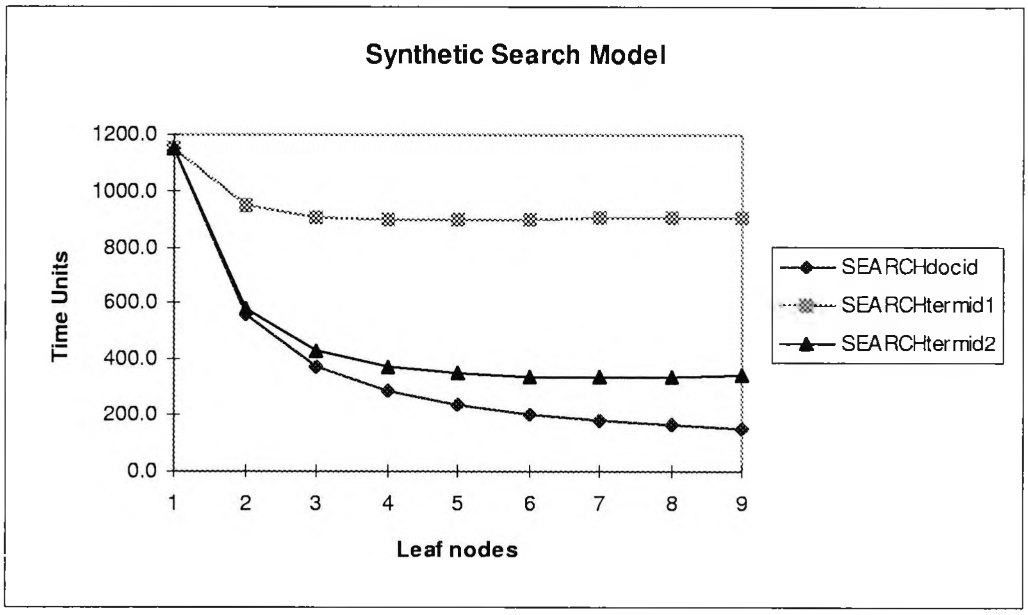


Fig 5-3. Comparative results for search models on 1-9 leaf nodes

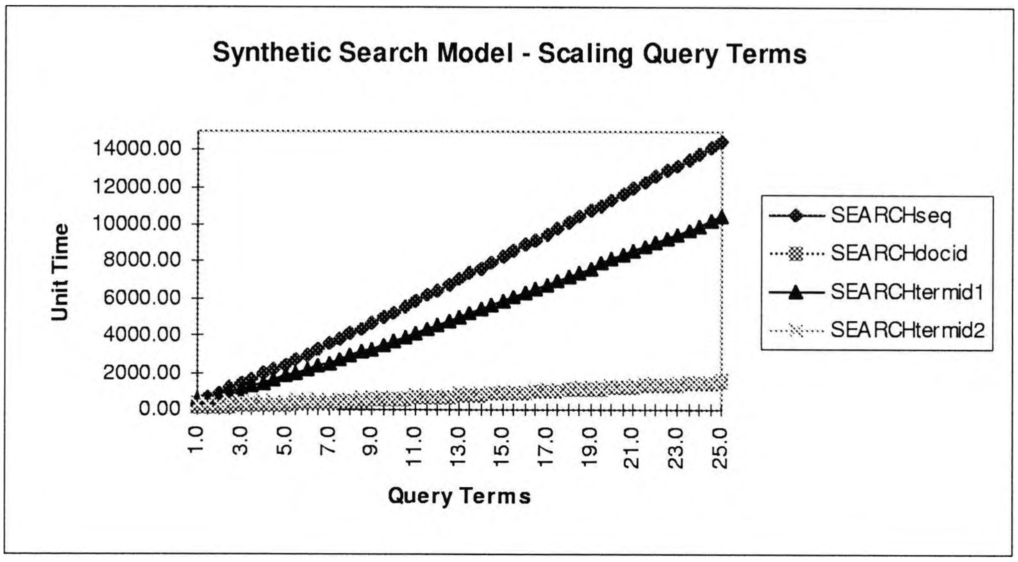


Fig 5-4a. Comparative results for search models: increasing query size.

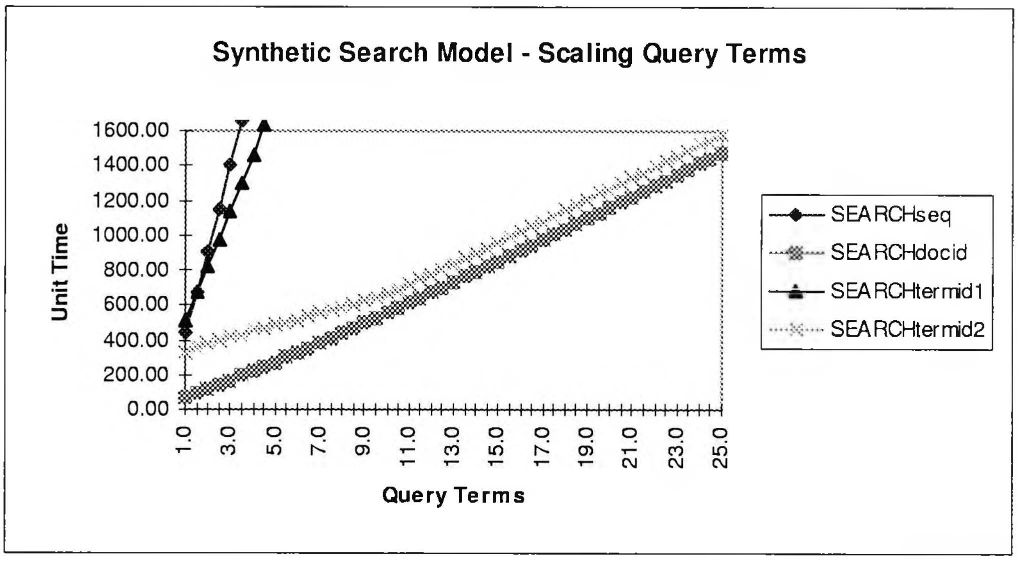


Fig 5-4b. Comparative results for search models: increasing query size (smaller scale which shows difference between *DocId* and *TermId* theoretical models).

Fig 5-4a shows that in theory, all parallel methods would outperform the sequential method for larger query sizes or larger data sets: the number of leaf nodes (the variable P) is set at 10 for this model. The comparison confirms that *TermId* with a sequential sort would not perform as well as the other two algorithms studied. It is difficult to see any difference between *DocId* and *TermId* with parallel sort because of the scale in fig 5-4a, but looking at a smaller scale version of the data in fig 5-4b we can see a clear difference between both theoretical models.

5.4 MODELS FOR PASSAGE RETRIEVAL

a	:	Average number of text atoms inspected per document
PR	:	Documents to do passage retrieval on

Table 5-6. Variables and constants for passage retrieval models

Table 5-6 declares the variables and constants used in the passage retrieval models. The search models declared in section 5.3 above are reused here: a normal probabilistic search precedes the application of the passage retrieval algorithm. The algorithms modelled are described in detail in chapter 4, section 4.7. We assume that a number of optimisations are used to reduce the n^2 inspected passages to (a(a-1))/2.

5.4.1 *Description of passage retrieval models*

The total cost of passage retrieval search will include the sequential probabilistic search. The $\text{PASSAGE}_{\text{seq}}$ model is constructed by defining a compute passages model and re-using the sequential probabilistic search cost model (see appendix A5.5). The parallel models are constructed in the same way for a given partitioning method, using the relevant parallel probabilistic search cost model (see appendix A5.6). The local passage processing method (see chapter 4, sub-section 4.7.2.2) does not need the addition of communication cost models, unlike distributed passage processing (see chapter 4, sub-section 4.7.2.1). Distributed passage processing requires that the top PR documents be identified using probabilistic search. This top set of documents is communicated to processors in the parallel machine and collected up again when the passage computation is complete: both partitioning methods use this communication cost model. The *TermId* partitioning method has a further and considerable communication cost in order to exchange passage data: one communication invocation per passage.

5.4.2 *Comparative results using passage retrieval models*

Fig 5-5 shows results on 1 to 9 processors with the values set at a=11 and PR=1000. As with previous models these values were gathered from the BASE1 collection. There is clearly a significant problem with using *TermId* partitioning with the passage retrieval algorithm we study in this thesis. The predicted comparative performance is so poor that the *DocId* models appear to be near zero. The argument used in chapter 4, section 4.7.2 against using *TermId* partitioning for passage retrieval is considerably strengthened by these theoretical results. It is clear from the model that predicted communication costs would increase the run time of any parallel program using such a partitioning method.

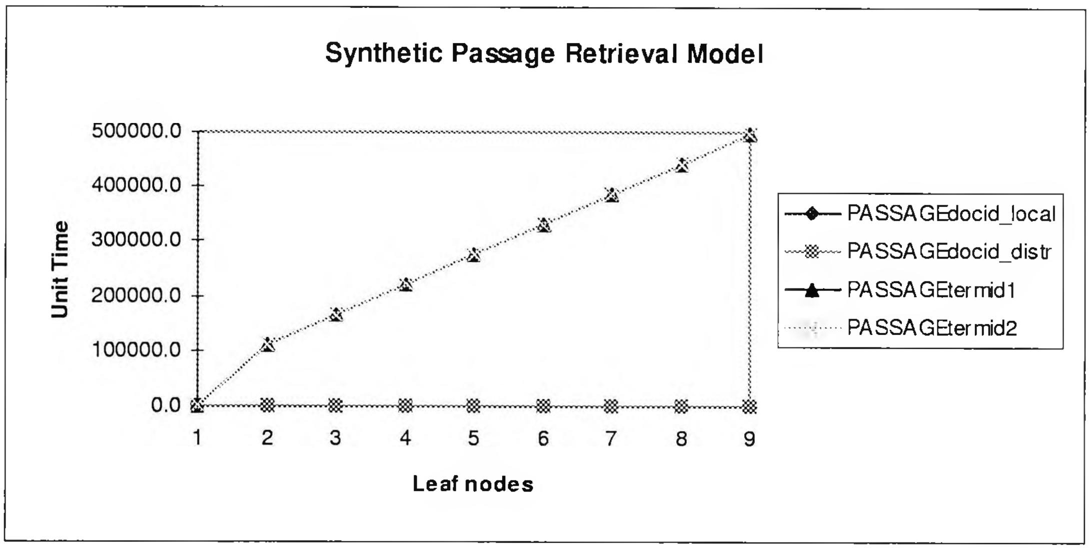


Fig 5-5. Synthetic passage retrieval model on 1 to 9 leaf nodes

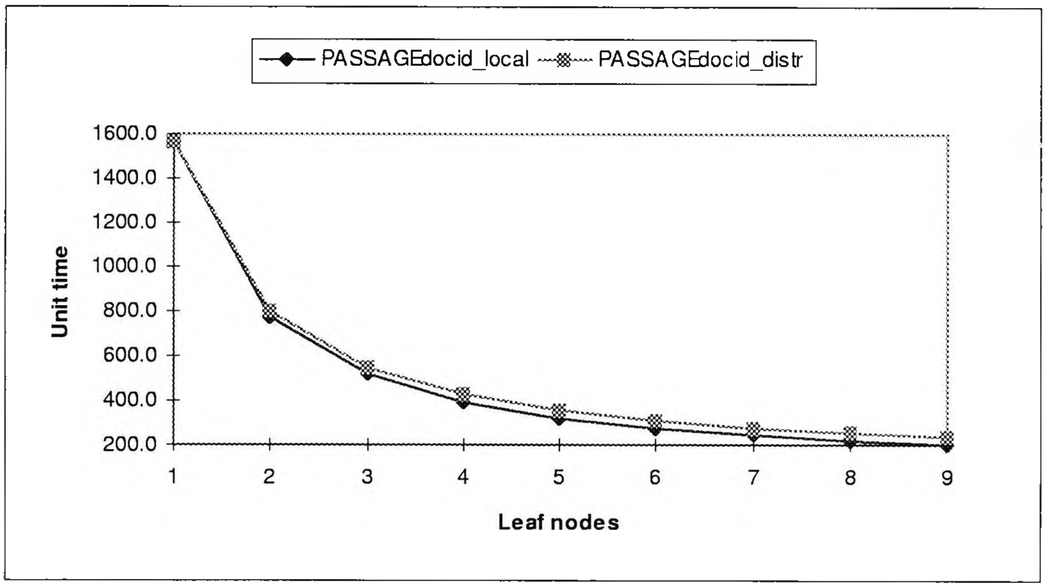


Fig 5-6. Synthetic passage retrieval models - *DocId* only

Fig 5-6 shows synthetic model results for *DocId* partitioning only. With both types of passage retrieval using *DocId* the prediction is time reduction with increasing numbers of processors, but the local method (marked $\text{PASSAGE}_{\text{docid\_local}}$) shows slightly better theoretical results than the distributed method (marked $\text{PASSAGE}_{\text{docid\_distr}}$).

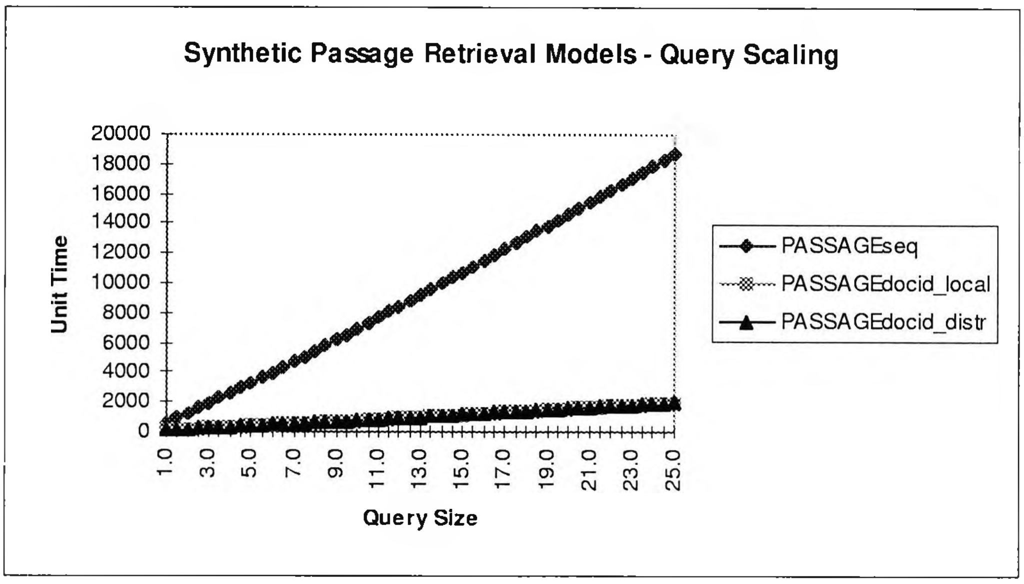


Fig 5-7. Synthetic passage retrieval models - Query Scaling 1 to 25 terms

Fig 5-7 shows the comparison between the type of build on *DocId* partitioning and sequential models only, varying the number of terms in the query. The number of leaf nodes (the variable P is set at 10). The synthetic models predict a considerably better performance for the parallel algorithms over the sequential algorithm. Due to the scaling it is difficult to perceive any difference between the two parallel methods.

5.5 TERM SELECTION MODELS FOR ROUTING/FILTERING

In this section we only deal with one algorithm using one term operation, otherwise we have to define 90 synthetic models (3 algorithms * 3 operations * 5 data distribution methods and sequential method * 2 reweighting schemes). We restrict the number of models to derive and inspect, reasoning that a distribution method which does not work in one model will not work in another. We say this because all algorithms use the same iterative technique (see chapter 4, section 4.8). We define models for *find best* with *add only* or *add reweight* operation: this requires the definition of ten models. Table 5-7 shows the variables and constants used for term selection models. We reuse many of the variables declared for search in table 5.4. We limit the maximum number of iterations to 100 (we used this limit in our experiments to be described in chapter 10). A further optimisation is used in our experiments is not to examine a term for ten iterations if that term has failed to improve the score in four iterations (see chapter 10, section 10.2). We simplify the modelling of this by using the u variable which is a percentage of the total examined in the term selection process.

r:	Number of documents judged relevant.
q:	Number of terms in term selection process: e.g. no of query terms (declared in table 5-4).
i:	Number of iterations in term selection.
u:	Estimate of total terms skipped during term selection (percentage of total terms examined).
w:	Maximum number of times a term is reweighted.
s:	Average set size (declared in table 5-4).
R[q,s]:	Final result set size lookup (declared in table 5-4).

Table 5-7. Variables and constants for term selection models

5.5.1 *Description of term selection models*

The modelling of term selection starts with the examination of evaluation costs. An evaluation of a term consists of applying the term to a training set and producing a score which is used to examine its fitness (see sub-section 4.8.2.1, chapter 4). A function for the cost of evaluation is constructed and declared in appendix A5.7.1. The number of terms selected for evaluation will depend on the type of term selection algorithm used, together with the term operation utilised (see section 4.8.2.2, chapter 4). We restrict our study to the *find best* algorithm together with *add only* and *add reweight* operations. We define a cost function for terms inspected (see appendix A5.7.2). Data for the query to be optimised must be loaded into memory either at the start of the optimisation process or when data is needed for a given term. We take the former approach assuming that the machine has enough in-core or virtual memory to be able to store the data (see appendix A5.7.3). The sequential models for routing (ROUTING_{seq} and ROUTING_{seqw}) are formed by multiplying the evaluations by the inspected terms and adding the load cost function (see appendix A5.7.4). The add reweight routing cost function (ROUTING_{seqw}) is further refined by multiplying the evaluation/inspected terms cost with the reweighting variable 'w': this is applicable to all models, sequential and parallel.

The basic parallel term selection models (ROUTING_{par} and ROUTING_{parw}) with no synchronisation or communication costs can be constructed by taking the sequential models, dividing them by the number of processors P, and then multiplying the result by the estimate of load imbalance LI[P] (see appendix A5.8). These basic models can be used to form cost models for *TermId* partitioning (ROUTING_{termid}), and replication (ROUTING_{rep}) and their reweighing counterparts. Models for the other two data distribution methods under consideration, *On-the-fly* distribution (ROUTING_{parfly} and $\text{ROUTING}_{parflyw}$) and *DocId* partitioning (ROUTING_{docid} and ROUTING_{docidw}) must be defined without using those basic models. In the model for *On-the-fly* distribution, data must be loaded centrally and is a sequential bottleneck. Inter-set parallelism cannot be sensibly used in *DocId* partitioning (see chapter 4, sub-section 4.8.3.5). We can have a distributed accumulated set and parallelism on merges and sorts so there is potential benefit: but only one evaluation is done at a time (parallel evaluations are done - intra-set parallelism). All models need to take account of synchronisation costs at the end of each outer iteration, which includes the merge of chosen term data into the accumulated *document identifier* set (see chapter 4, sub-section 4.8.2.1).

The communications overheads are modelled differently for each parallel routing scheme. Costs for *DocId* partitioning will be higher as communication is needed for inner iterations as well as outer ones. The merge at the synchronisation point is much cheaper as it could be done in parallel: all other models require a sequential merge at the same point. The interaction for *TermId* partitioning would require the retrieval of a terms *document identifier* set from one node in order to be broadcast to other processors. Communication costs at the synchronisation point for *On-the-fly* distribution would be much less as the master process has direct access to all data, but would be more expensive for the initial load as all term data must be broadcast to the processors. In *replication*, we assume that the main problem would be latency when exchanging set identifier data at the synchronisation point.

r:	50
n:	300
u:	0.4
w:	3

Table 5-8. Values used for term selection models

5.5.2 *Comparative results using term selection models*

Table 5-8 shows the values used for the theoretical results produced in this section. The values for resource cost variables (T_{cpu}, T_{trans}, T_{seek}, T_{com}) are taken from table 5-5. The values chosen for variables declared in table 5-8 are taken in the main from Ziff-Davis experiments described in chapter 10, apart from the variable r. Fig 5-8 shows the comparison between models with the number of processors P set at 100 and the number of iterations i set at 100.

The models predict that the best performing distribution scheme overall would be the *replication* method. On smaller processor sets, the *DocId* partitioning method shows better theoretical results, but predicts that the performance would deteriorate substantially due the restriction on the level of parallelism in that distribution method: the communication/ computation balance would be skewed. The prediction with respect to *TermId* partitioning and *on-the-fly* distribution is that there is little difference between them particularly with large processor sets. Fig 5-9 shows theoretical results varying the number of query terms in the selection process (the variable n). The number of slave nodes (P) for parallel models is set at 10. The models predict that all parallel methods would outperform the sequential or uniprocessor method and that the difference would increase proportionally with the number of terms in the query. Fig 5-10 shows theoretical results for parallel models only on the same data.

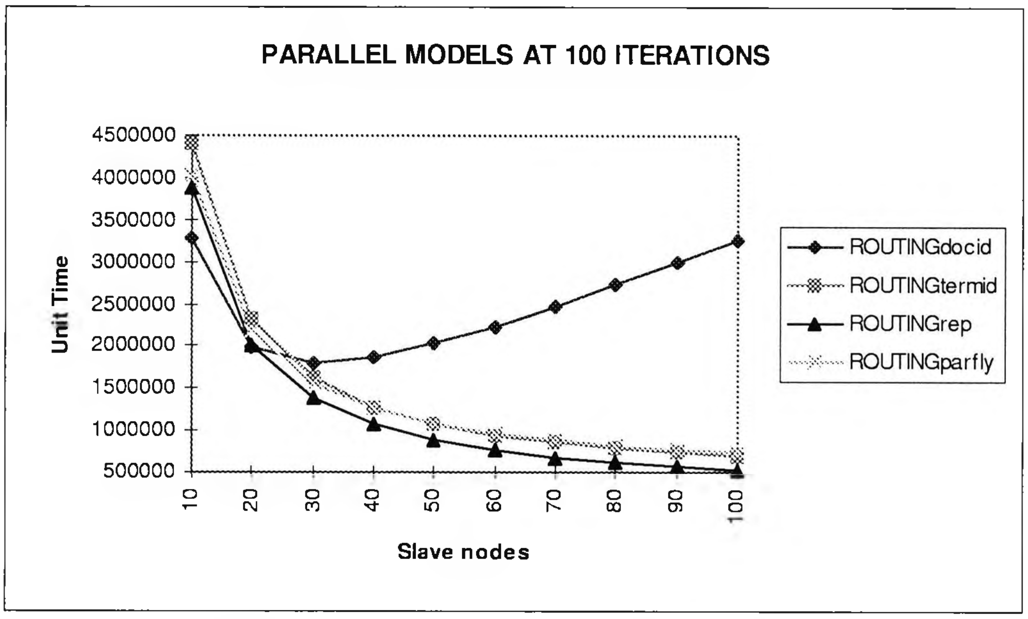


Fig 5-8. Term selection model results using large slave node set and iteration sizes

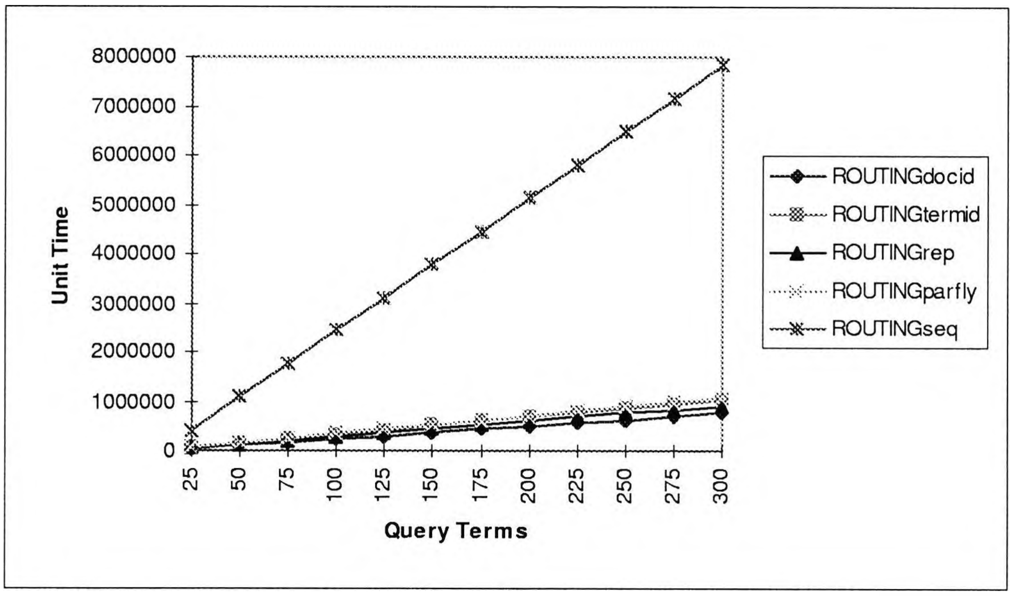


Fig 5-9. Comparison between all models by varying number of query terms

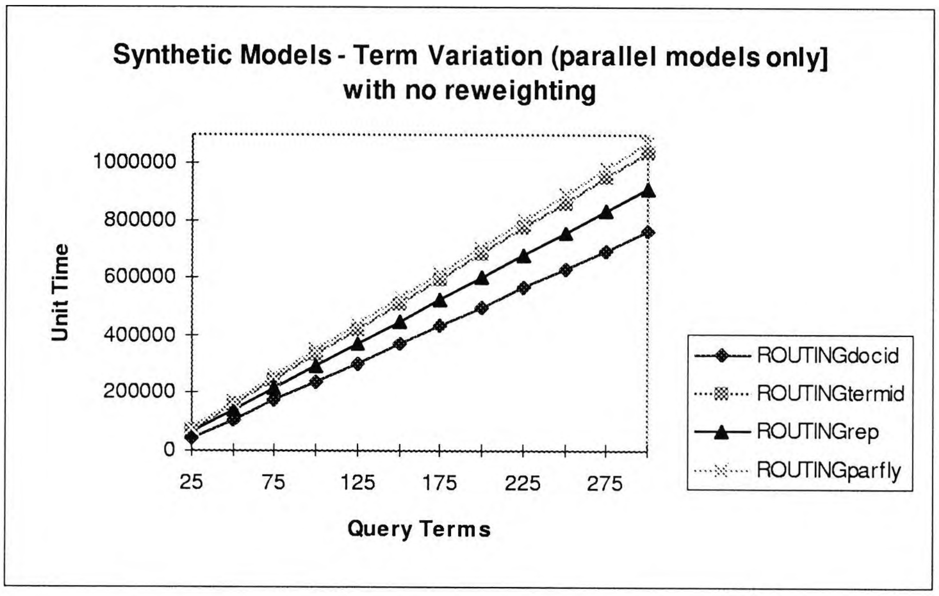


Fig 5-10. Comparison between parallel models by varying number of query terms

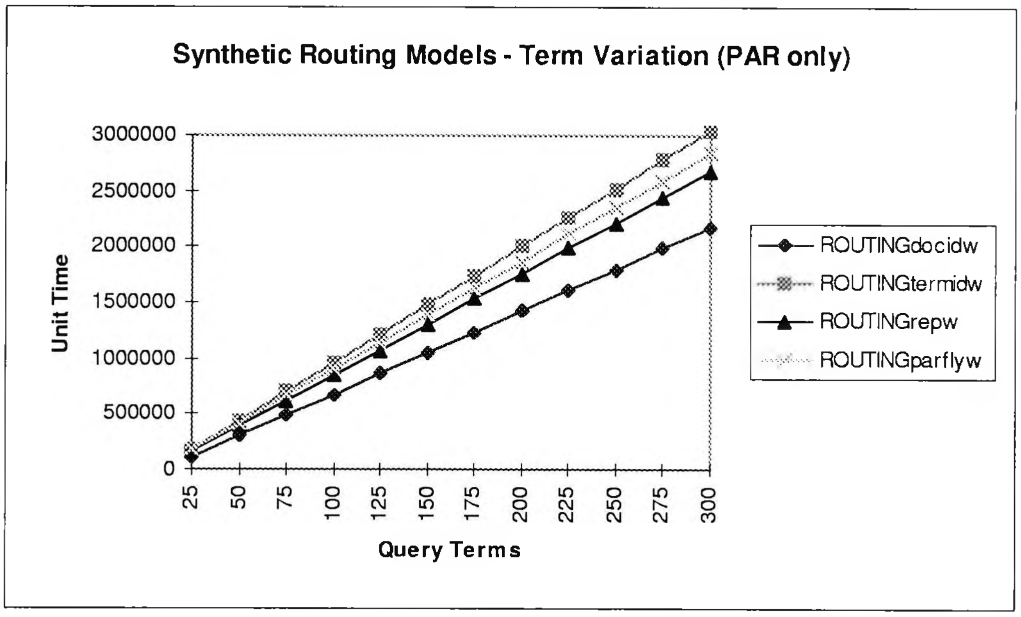


Fig 5-11. Comparison between parallel models by varying number of query terms
(weight variation)

The number of iterations (i) is set at 20. The theoretical results given in fig 5-10 are consistent with those in fig 5-8, namely that *DocId* partitioning may perform better on smaller numbers of processors and that there is little difference between *TermId* partitioning and *on-the-fly* distribution (although *TermId* may perform slightly better). However if we use a weight variation model the prediction is that *on-the-fly* distribution may perform slightly better (see fig 5-11). The number of iterations (i) for these models is set at 30 to reflect the extra workload weight variation places on the term selection process (found in experiments to be described in chapter 10).

5.6 MODELS FOR INDEX UPDATE

Table 5-9 shows variables and constants for synthetic index update models. We use a contention factor (c) to simplify the modelling as it is difficult to establish a relationship between concurrent index update and transaction service (even with the strict interleaving method used for the purposes of this thesis). We assume an increase in contention with more processors. We examine several different aspects of index update in this section. The comparative performance of transactions on partitioning methods is studied when transactions only are serviced and when transactions contend with index reorganisation for resources. We also look at the comparative performance on the index reorganisation with and without the contention factor. We vary the contention factor in order to study the effect on transactions and index reorganisation. The algorithms and methods discussed are those described in chapter 4, section 4.6.

dict:	Size of the dictionary.
ur :	Update rate (number of updates in time period).
sr :	Search rate (number of search requests in time period).
ro :	Percentage of total transaction time spent re-organising database.
b:	Blocking factor for dictionary file.
c[P]:	Contention for resources factor at P processors
m:	Number of keywords in the buffer on index update.
t:	Number of keywords transferred during index update (total words handled = m+t).
i[P]:	Increase in number of terms for index for P processors [*DocId* only].
p[P]:	Percentage decrease in average set size at P processors [*DocId* only].

Table 5-9. Variables and constants for index update

5.6.1 *Description of index update models*

In our sequential models we assume a Client/Server model is used to service transactions on a single unpartitioned inverted file (see appendix A5.9). We construct cost models for updates (see appendix A5.9.1) and use these and the search models defined to create models with contention using the c[P] variable (see appendix A5.9.2). These functions allow us to define a transaction cost function in which we can vary the update to search ratio and the contention rate (see appendix A5.9.3). We take the same strategy with the parallel transaction models (see appendices A5.10.1 and A5.10.2). In the *TermId* transaction model we specify the parallel sort search cost model.

When a buffer has reached a given size, an index update or reorganisation is done which makes changes to the index persistent (see chapter 4, sub-section 4.6.1.2). The algorithm consists of examing the in-core data, identifying keywords and merging data, together with reading in and writing data to disk. The sequential cost model for this is defined in appendix A5.9.4. The contention model for reorganising the index is constructed simply by adding the estimated cost on one node. The models for *TermId* partitioning can be formed by applying parallel variables to the models defined in appendix A5.9.4 (see appendix A5.10.4). We cannot directly use the sequential functions defined in appendix A5.9.4 for *DocId* partitioning (see appendix A5.10.3). This is for three main reasons. Firstly the I/O activity is different (more seek cost, reduced transfer times). The other two related to the way the partitioning method interacts with the available processors in the system. The dictionary will be larger as terms are replicated across partitions, but the average set size to be reorganised decreases as hit terms will be dispersed in blocks with less frequent terms.

5.6.2 *Comparative results using index update model: Transaction Processing*

Table 5-10 shows the variables and constants used for our transaction processing comparisons. The values for resource cost variables (T_{cpu}, T_{trans}, T_{seek}, T_{com}) are taken from table 5-5. The *dict* and *s* variables are an estimate taken from the BASE1 collection. We assume that documents to be inserted are small (we use short document for insertions in chapter 8).

dict	:	1,355,140
ur	:	10
sr	:	100
ro	:	0 to 100%
b	:	10
c[P]	:	0.05 at with an accumulated increase of 0.25 per processor.
s	:	7791
n	:	50

Table 5-10. Variables and constants values used for update models

Fig 5-12 shows the prediction using these values and compares the theoretical performance between both types of partitioning methods where transactions are affected by index update (models are labelled with the suffix RO) and normal transaction processing without contention for resources. The models predict that *DocId* partitioning will outperform

TermId whether or not there is contention for resources due to index update. As you would expect, where there is contention for resources the predicted performance is worse than non-contention models. The model on *TermId* partitioning in the presence of an index update predicts a deterioration in performance with increasing numbers of processors: with *DocId* the prediction is that performance will remain constant after a certain number of processors is reached.

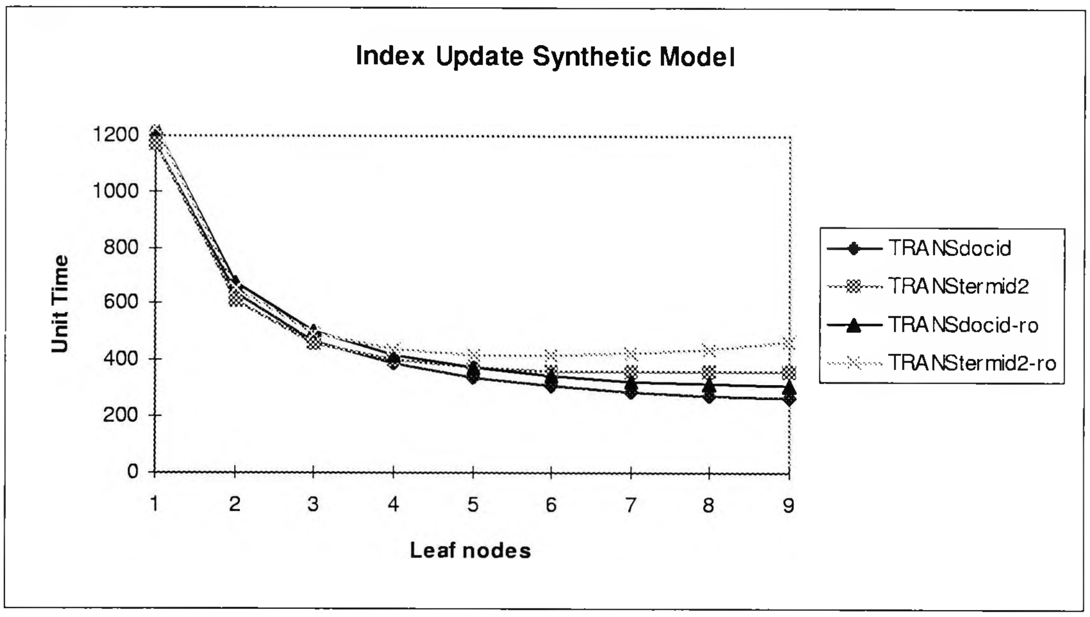


Fig 5-12. Comparison between partitioning methods in presence and absence of index update

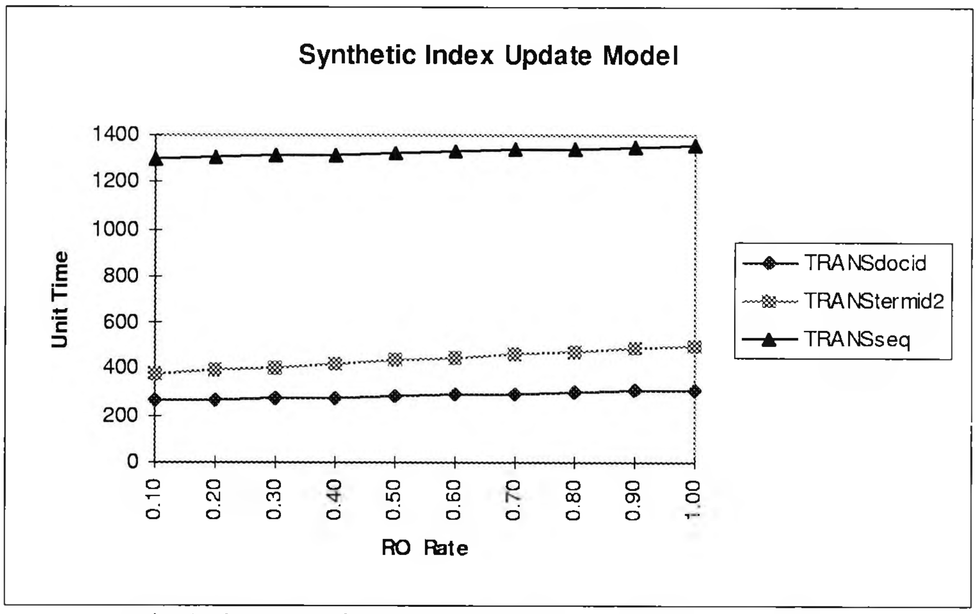


Fig 5-13. Comparison between models: variation of ro factor and its effect on transaction processing.

Figs 5-13 shows the effect in theory of varying the time spent reorganising the index on both partitioning methods and a theoretical uniprocessor run. The number of processors (P) for parallel models is fixed at 10. The balance between transactions is set at 100 queries or searches per 10 update transactions. Again the comparison predicts that *DocId* would outperform *TermId* partitioning and that they both would both outperform a uniprocessor. There is an increase in unit time for all models but the increase is only very slight.

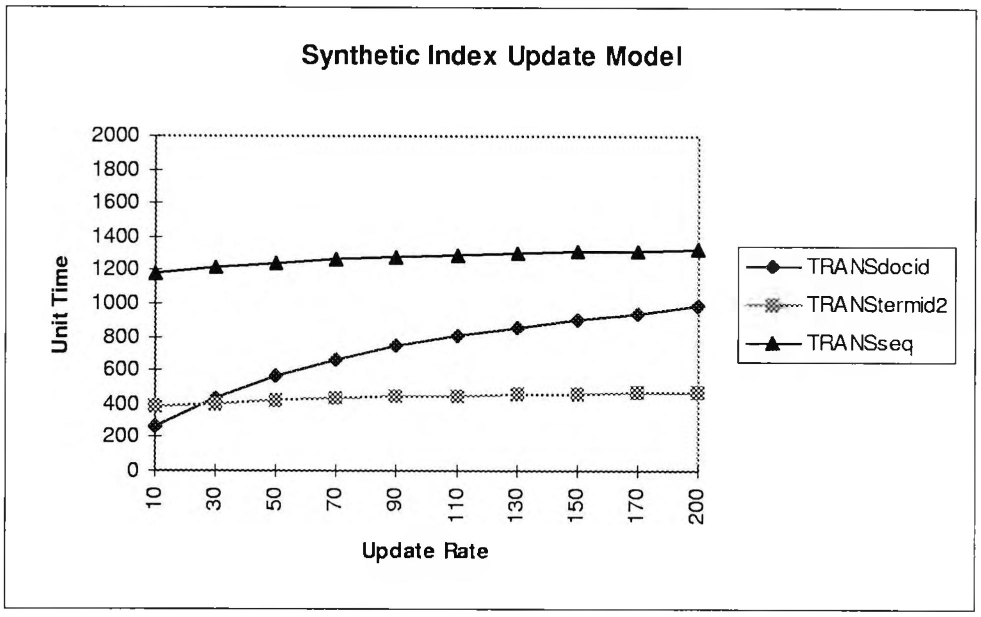


Fig 5-14. Comparison between models: variation of update rate and its effect on transaction processing.

Fig 5-14 shows the effect of varying the update rate and keeping the ro rate constant (the rate is set at 0.1 in the graph, but changing the variable to another value does not affect the relative position of the models on the graph). The number of processors (P) is set at 10 and the search/query rate is set at 100. The prediction is that both parallel methods would outperform the uniprocessor method and that *TermId* would be the better performing parallel method overall: the parallelism available in some aspects of updating *TermId* indexes is beneficial in theory. However the models also predict that *DocId* would converge towards the uniprocessor model with an increasing update rate: the reason for this is that updates would dominate the overall transaction time to the detriment of *DocId* performance.

5.6.3 *Comparative results using index update model: index reorganisation*

Fig 5-15 shows the effect of varying the number of processors on the theoretical models. The key prediction here is that the extra seek time which *DocId* partitioning will place on reorganisation of the index will not adversely affect the theoretical improvement in performance by deploying parallelism with that method. This is because the amount of data actually moved per term block will reduce with increasing the number of partitions with *DocId*: the prediction is that *DocId* partitioning will outperform *TermId* in this aspect of index reorganisation.

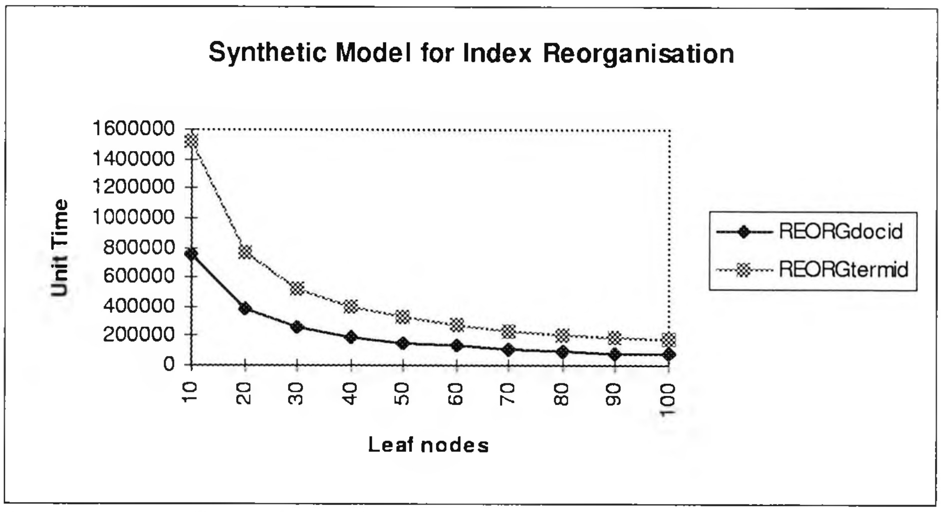


Fig 5-15. Comparison between partitioning methods: varying parallel machine size

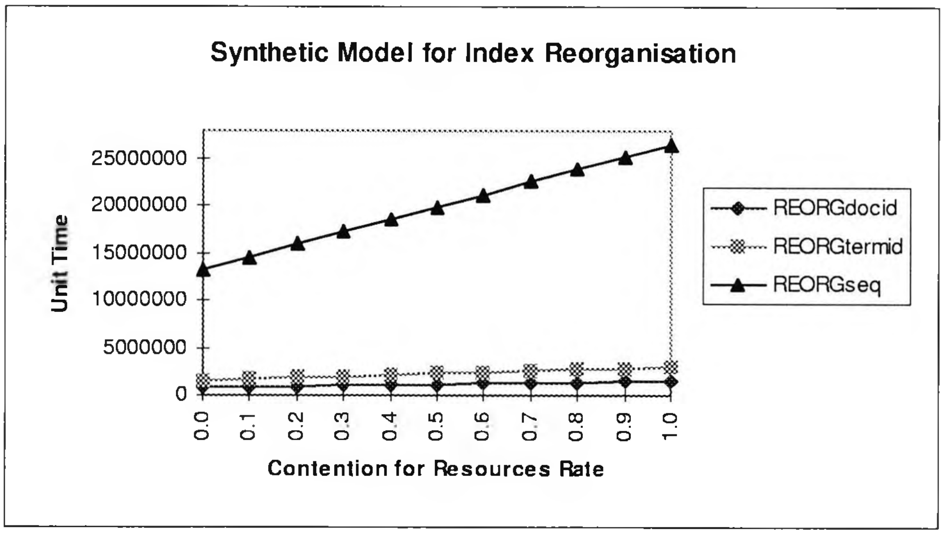


Fig 5-16. Effect of contention for resources on theoretical models

Fig 5-16 show the effect of resource contention on the models. The number of processors (P) set for the parallel machine is 10. In theory the parallel models are much better able to cope with resource contention, and of the two parallel models *DocId* partitioning is predicted to be the best. The theoretical run time for both parallel models does increase but only slightly compared to the uniprocessor model (labelled $\text{REORG}_{\text{seq}}$ in the diagram). This prediction is due in most part to the amount of data moved for *DocId* index reorganisations. The same settings used in contention predictions are used in fig 5-17 which shows the effect of buffer size on the models. The theoretical prediction is the same here as it was with contention in the models and for the same reasons: namely *DocId* partitioning is predicted to be the best and both parallel models predict that performance with large buffers would be better than uniprocessor operation.

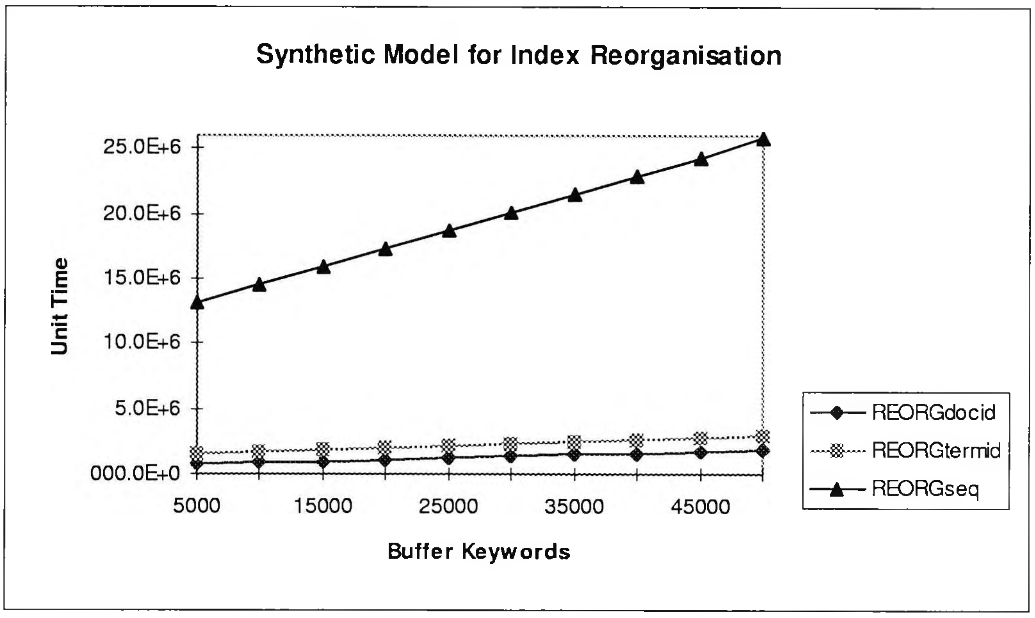


Fig 5-17. Effect of buffer size on theoretical models

5.7 DISCUSSION OF SYNTHETIC MODEL RESULTS

The synthetic models produced in this chapter predict that for most tasks the *DocId* partitioning method would be the better performing data distribution scheme of those studied. For the index, probabilistic search, passage retrieval and index update tasks the prediction is unambiguous. For the passage retrieval task in particular it is very clear that the *TermId* partitioning method is simply not viable: we therefore intend to study the use of *DocId* partitioning only for this task using both types of passage processing, local and distributed. For the other tasks apart from routing/filtering we do experiments on the partitioning methods to compare and contrast them.

The theoretical evidence on the routing/filtering task is more complicated and therefore needs more discussion. On small numbers of processors the prediction is that *DocId* partitioning would be the best data distribution scheme, but on larger parallel machines the performance would deteriorate due to excessive communication. Due to the restrictions on inter-set parallelism, the proposed method for *DocId* partitioning does not show the same promise as intra-set parallelism usable on the other distribution methods. Of the other three, the prediction is that *replication* would be the best performing method and that performance with *on-the-fly* distribution would be about the same with *TermId* partitioning. We therefore concentrate our efforts on *replication* and *on-the-fly* distribution schemes, partly because of the theoretical comparison and partly because such methods are easier to manage (we only need to initiate one indexing run to be able to do experiments, whereas with *TermId* partitioning we must initiate an indexing for every node set used).

In our empirical examination of the information retrieval tasks under discussion we examine how good these theoretical predictions are; for comparison purposes (are the best performing data distribution schemes in theory best in practice), relative difference between the data distribution methods and to see if any generic statements about parallelism in these tasks can be made.

Chapter 6

Indexing Results

6.1 INTRODUCTION

The generation of inverted indexes for text databases is a computationally intensive process that requires the exclusive use of processing resources for long periods. This chapter describes results using the indexing techniques described in chapter 4, section 4.4. We give results for both *TermId* and *DocId* partitioning methods. The hardware used for these experiments is AP3000, Alpha Farm and Pentium Cluster described in chapter 3, section 3.2. The main bulk of the chapter describes our results on the AP3000 and Alpha farm as applied to the BASE1, BASE10 and BASE10 subsets (BASE2, BASE4, BASE6 and BASE8) both in terms of time (section 6.3) and space costs (section 6.4). Details of the collections can be found in chapter 3, section 3.3.1. We describe experimental details for both of these sections in section 6.2. We then describe the experiments done at TREC8 (MacFarlane et al, 2000a) in section 6.5 using the best performing method found in the main experiments. The TREC8 experiments were done on Ad-Hoc data as well as full web data used in the web track: the Pentium cluster was utilised. We conclude in section 6.6 by comparing and contrasting the results.

6.2 DATA DISTRIBUTION AND EXPERIMENT DETAILS

We used both types of build described in chapter 4, section 4.4 namely *local* and *distributed* builds. The strategy used to distribute the BASE1 and BASE10 collections for *local build* was to evenly spread the directories (which contain the files for these collections) among the nodes as far as possible. The requirement of a distribution strategy is to get the best possible load balance for indexing as well as probabilistic and passage retrieval search. The distribution process was done before the indexing program was started, and is not included in the timings. The *local build* experiments were run on eight nodes of both the Alpha Farm and the AP3000. For *distributed builds* we record runs on one to seven leaves for both types of partitioning method. We record runs on inverted files both with and without position data. The document map described in chapter 3, section 3.2.4, was fragmented with *local build* and replicated with *distributed build.* Map data on *distributed build* with *DocId* could be fragmented, but we chose to replicate rather than maintain extra source code in order to save time.

6.3 INDEX GENERATION TIME COSTS AND PARTITIONING COMPARISON

In this section we declare the timing results on indexing using the configurations described above. The results are compared and contrasted where necessary as well as comparing them with available results for other systems on the BASE1 and BASE10 collections described in chapter 3, section 3.3.1. The measures discussed are: indexing elapsed time in hours, throughput, scalability, scaleup, speedup and efficiency, load imbalance (LI) and merging costs. Metrics used are defined in chapter 3, section 3.4.1.

6.3.1 *Indexing Elapsed Time*

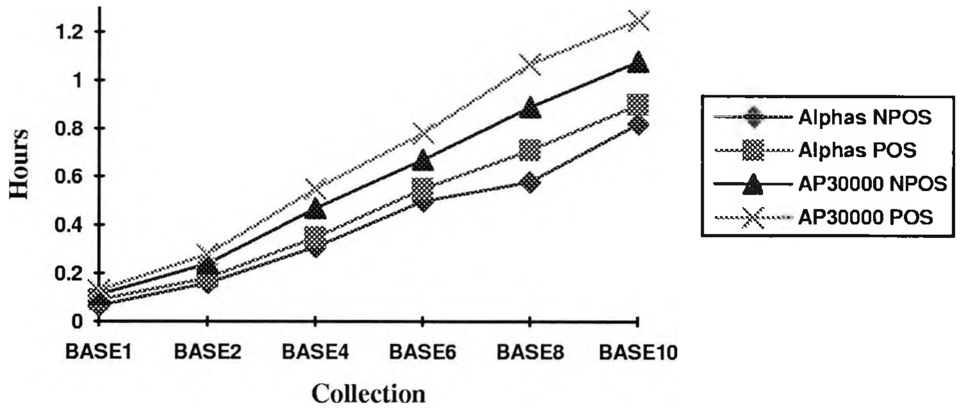


Fig 6-1. BASE1-10 *local build* [*DocId*]: indexing elapsed time in hours

In general, the Alpha farm was much faster than the AP3000 for indexing elapsed time as its processors are faster. For example on BASE10 *local build* indexing with postings only data took 0.82 hours on the Alphas and 1.08 hours on the AP3000 (see fig 6.1). The Alpha elapsed times recorded on *local build* also compare well with the results given at VLC2 (Hawking et al, 1999). That is, on BASE1 only two groups report slightly faster times than our posting only elapsed time of 0.065 hours (0.043 and 0.052 hours). Our sequential elapsed time on BASE1 at 0.56 (postings only) also compares well with those groups utilising a single processor: two other groups using uniprocessors recorded 0.42 and 1 hour respectively (refer to figs 6-2 and 6-3). On BASE10 on the Alphas the comparison is even more encouraging: only one group records a faster time of 0.504 hours. It should be noted that while the group with the fastest BASE10 indexing time uses a much smaller machine configuration (4 Intel PII processors) they use a very different method of inversion in which the collection is treated as one document (Clarke et al, 1998).

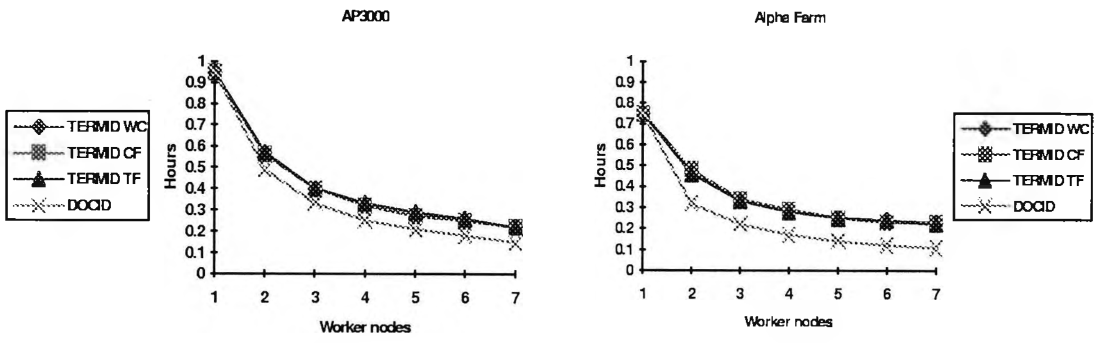


Fig 6-2. BASE1 *distributed build*: indexing elapsed times in hours (position data)*
(*Note: refer back to section 4.4.2 re: WC/CF/TF)

The results for *distributed build* indexing are presented in figs 6.2 and 6.3. The elapsed times for *DocId* are much better than those for the *TermId* method. This trend can be seen in all of the diagrams irrespective of machine or inverted file type used. The smallest difference is found on indexes with postings only using the AP3000. In general *TermId* elapsed times were longer than *DocId* because of the amount of data that has to be exchanged between nodes for the method, particularly for indexes with position data. Very little difference in time was found in any of the term allocation strategies (see section 4.4.2) studied for *TermId.*

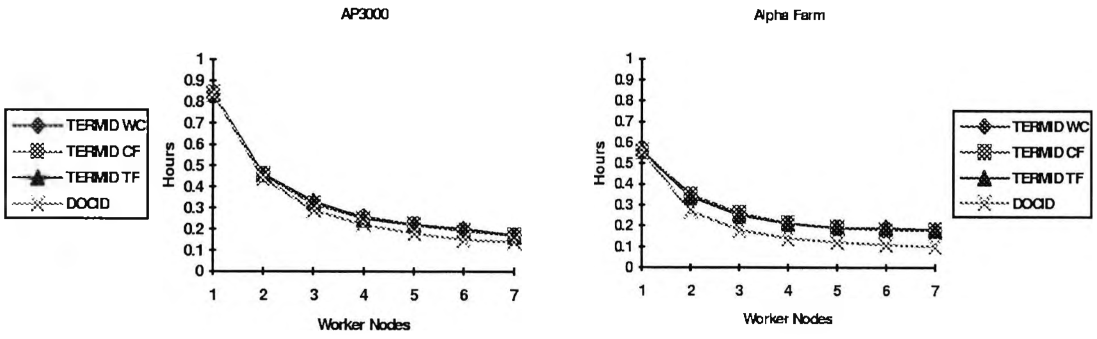


Fig 6-3. BASE1 *distributed build* : indexing elapsed times in hours (postings only)

One interesting factor found in the *TermId* results was that the AP3000 outperformed the Alpha farm at 7 worker nodes largely due to the extra network bandwidth available. It is that this point where the compute/communication balance favours the AP3000. A further run using *distributed build* with *DocId* partitioning on the Alpha farm revealed how much faster it is to use the ATM network than the Ethernet network: the time with ATM on 2 worker nodes building an index for BASE1 with no position data was 0.27 hours, while the figure for

Ethernet was nearly double at 0.47 hours. This comparison further illustrates the importance of network bandwidth to the *distributed build* method and which can cause problems in many IR tasks (Rungsawang et al, 1999). We did not conduct any further experiments on this type of build for indexing using the Ethernet network as a consequence.

The extra time costs engendered by generating inversion with position data varied (this ratio is declared in chapter 3, sub-section 3.4.1.2 - our aim is to record a ratio as close to 1.0 as possible). For example, in *local build DocId* the difference between posting only generation and position data generation ranged between 1.09 - 1.37 times on the Alphas (where merging was required). The extra costs on BASE1 are the highest (1.25 for the AP3000 and 1.37 for the Alphas) because the index with postings only is saved directly to disk without the need for merging: merging is required only when memory limits have been exceeded. Fig 6.4 shows the ratios for *distributed build* experiments. How much these extra costs are justified depends on the query processing requirement: such as a user need for passage retrieval or proximity operators.

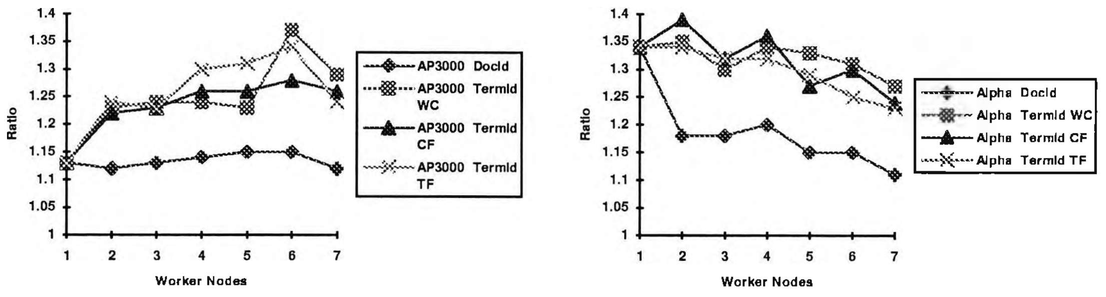


Fig 6-4. BASE1 *distributed build*: indexing extra costs for storage of position data

6.3.2 *Throughput*

The metric we use for throughput is Gigabytes of text processed per hour (G/Hour) to compare performance between database builds. Fig 6.5 shows the throughput for 8 processor configurations. The throughput for the Alphas is much faster than for the AP3000, e.g. on BASE1 *local build* indexing with postings only the rate is 15.4 G/Hour compared with 9.5 G/Hour on the AP3000. These are by far the best throughput results because no merging was needed: the configuration had enough memory to store the whole index and save it directly. The rate for other collections for *local build* indexing was 12-14 G/Hour on the Alphas for postings only. Only one VLC2 participant recorded faster throughput for BASE1 and BASE10 collections (just over 19 G/Hour). The throughput on BASE1 using *distributed build DocId* with is not as good the *local build* but is still encouraging (see fig 6.6).

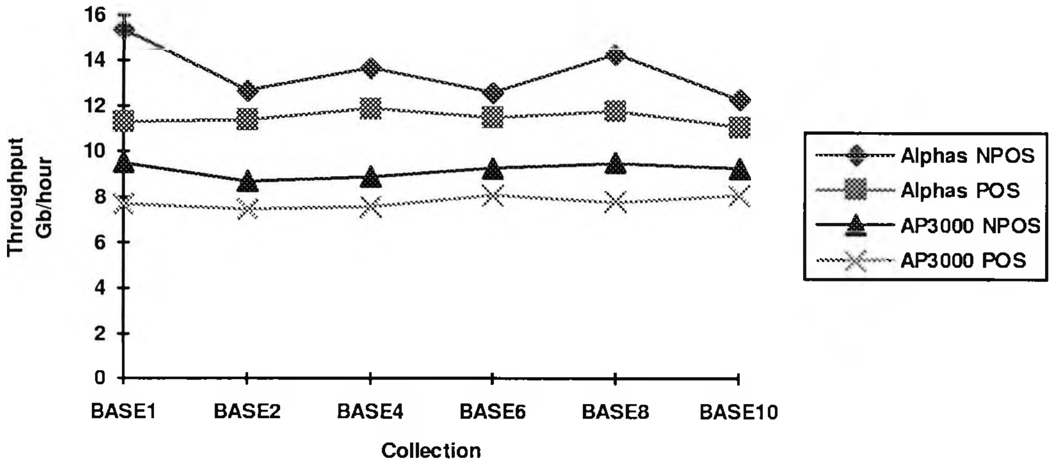


Fig 6-5. BASE1-BASE10 *local build* [*DocId*]: indexing Gb/Hour throughput

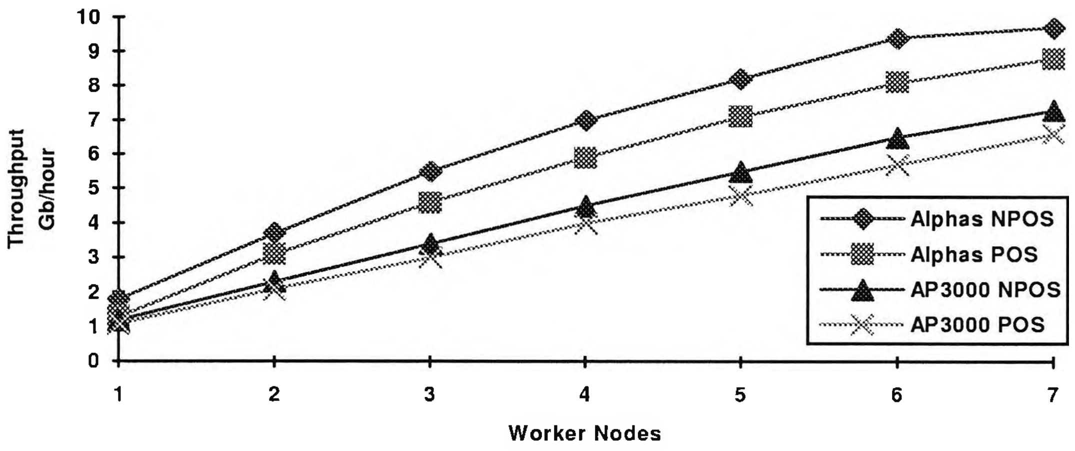


Fig 6-6. BASE1 *distributed build* [*DocId*]: indexing Gb/Hour throughput

It was found that increasing the number of worker nodes increased the throughput for both *distributed build* methods. For example, the *DocId* results for 7 worker nodes yielded a throughput of 9.7 G/Hour on the Alphas for postings only data indexes, compared with 1.8 for the uniprocessor experiment. The throughput for *TermId* builds was not as impressive but still acceptable with postings only: for example 5.8 G/Hour was recorded on the AP3000. The throughput for builds with position data was not as good, with 4.5 G/Hour on the AP3000 (see fig 6-7). Note that we only declare results for *TermId* with the word count (WC) method as there is very little difference in measurement between any of the term allocation strategies studied. Note also the superior performance in throughput on the AP3000 at 7 worker nodes due to the extra bandwidth available with that machine.

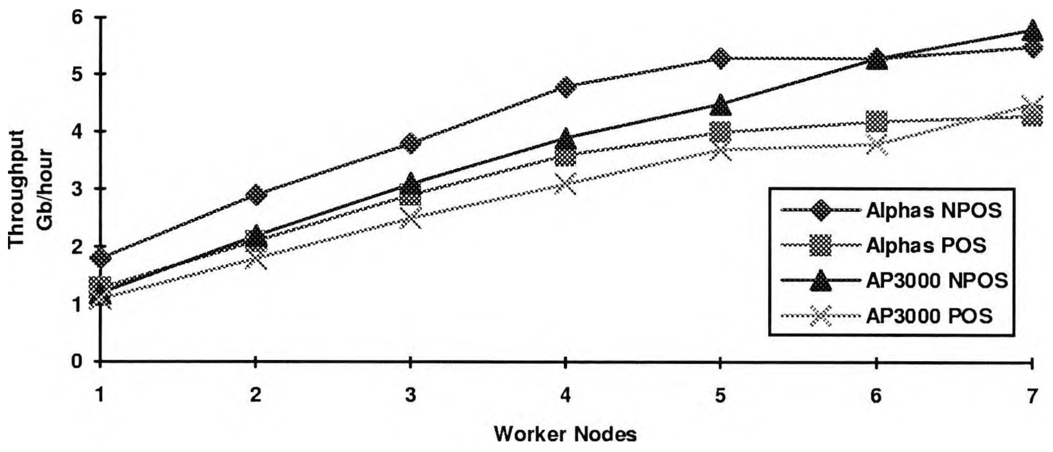


Fig 6-7. BASE1 *distributed build* [*TermId*]: indexing Gb/Hour throughput (WC only)

6.3.3 *Scalability*

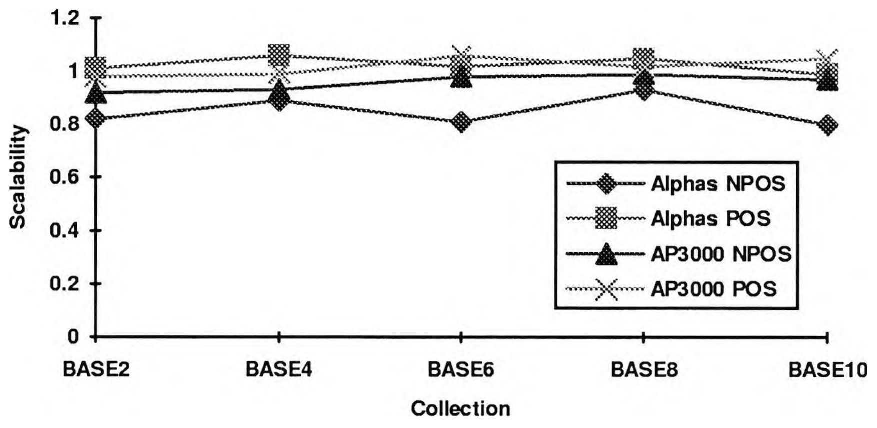


Fig 6-8. BASE2-BASE10 *local build* [DocId]: indexing scalability from BASE1

The data measure used in the equation is the size of indexed text. The scalability metric is defined in chapter 3, sub-section 3.4.1.6. We measure the effect of increasing collection size on the same sized parallel machine using the BASE2-10 collections over the BASE1 collection. We look for a scalability of around 1.0, greater than 1.0 being the aim. The results are presented in fig 6-8. With postings only data the scalability ranges between 0.80 and 0.93 on the Alphas and 0.92 and 0.99 on the AP3000. These figures are rather distorted because of the direct save on BASE1, that is no merging was needed as memory limits were not exceeded. The results are on the pessimistic side (if more memory was available we might be able to save indexes directly on all the collections studied). In builds with position data the scalability is excellent with the Alphas registering super-linear scalability on most BASEx (BASE10 was the exception) and the AP3000 delivering super-linear scalability on BASE6,8 and 10. The scalability results for indexes with position data demonstrate that the algorithms and data structures implemented are well able to cope with the extra computational load and data size that such builds both require and process.

6.3.4 *Scaleup*

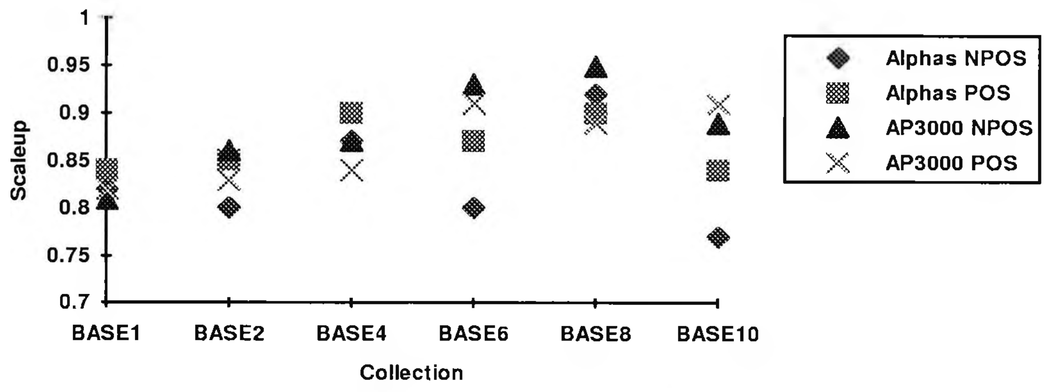


Fig 6-9. BASE1-BASE10 *local build* [*DocId*]: indexing scaleup

The scaleup metric is declared in chapter 3, sub-section 3.4.1.7. We measure within BASEx scaleup for *local build* only in this section. We take the times on each individual processor and compare the smallest elapsed time with the largest elapsed time on all 8 nodes. We are comparing the smallest sub-collection of BASEx (1/8th of BASEx) with the full sized BASEx collection. We use the least favourable figure in our measurement to obtain the lowest scaleup from any of the chosen sub-collections: our measurements are therefore pessimistic. We look for a scaleup of around 1.0, greater than 1.0 being the aim. The results are given in fig 6-9. In general the scaleups recorded are very good with most above the 0.8 mark. The worst scaleup was measured over the BASE10 collection on builds with no position data with a figure of 0.77. This figure was found on the Alpha farm where the processors are much faster. A combination of data size and processor speed can have an impact on scaleup: the scaleup figures for indexes with position data on the Alpha farm are generally superior to indexes without such data. The situation is reversed for AP3000 where the processors are slower. These scaleup figures show that there is little deterioration in performance of our implemented data structures and algorithms when moving from a smaller collection indexing on a small configuration parallel machine, compared with a larger collection on a larger configuration machine.

6.3.5 *Speedup and Efficiency*

All figures relate to the BASE1 collection. Definitions of these metrics can be found in chapter 3, sub-sections 3.4.1.4 and 3.4.1.5. Recall that our ideal speedup is equal to the number of nodes, whereas for efficiency we look for a figure of 1.0. A surprising feature was the superlinear speedup and efficiency figures found with some of the indexing experiments particularly for the *local build DocId* 8 processor runs (see table 6-1). For example with the direct save on postings only data *local build* on the Alphas yielded a speedup of 8.5 and efficiency of 1.07. This effect was also found on some of the runs using *Distributed DocId* indexing (see figs 6-10 and 6-11).

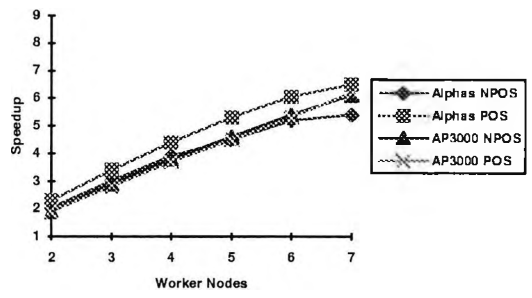


Fig 6-10. BASE1 *distributed build* [*DocId*]: indexing speedup

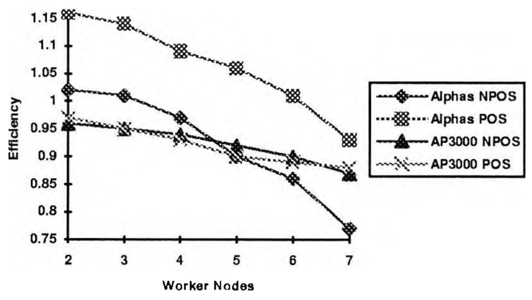


Fig 6-11. BASE1 *distributed build* [*DocId*]: indexing efficiency

Machine	File Type	Speedup	Efficiency
Alpha	NPOS	8.5	1.07
	POS	8.4	1.04
AP3000	NPOS	7.96	0.99
	POS	7.2	0.90

Table 6-1. BASE1 *local build* [*Docid*]: indexing speedup and efficiency

The reason this effect can occur is the extra memory multiple nodes have compared with a sequential processor, i.e. on *local build* with 8 nodes the index fits into main memory and it can be saved directly without the need for merging. More memory reduces the number of intermediate results saved to disk and therefore saves I/O time when data is merged to create the index. On *distributed build* a two *worker* configuration has twice the memory of the sequential program. The super-linear effect tails off at various stages on the *Distributed* version as communication time becomes more important (see fig 6.10).

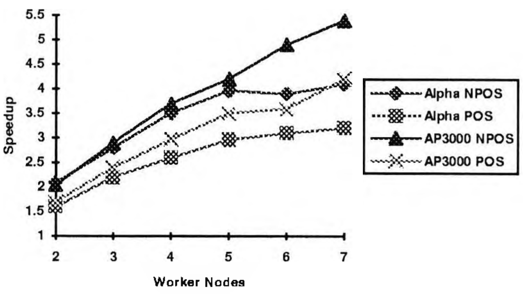


Fig 6-12. BASE1 *distributed build* [*TermId*]: indexing speedup (WC only)

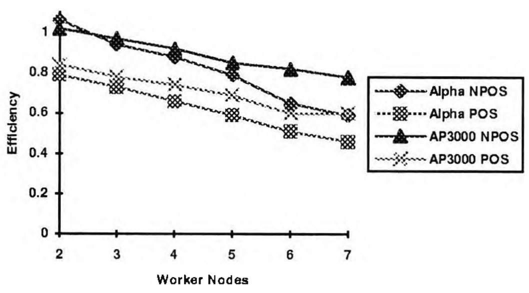


Fig 6-13. BASE1 *distributed build* [*TermId*]: indexing efficiency (WC only)

With *TermId* communication is very important: the global merge reduces most speedup/efficiency measures to less than linear (see figs 6.12 and 6.13). With position data and *TermId* there is little speedup on the Alpha Farm and efficiency ranges from the average to poor. Interestingly super-linear speedup/efficiency does occur on two worker nodes with builds on posting only data: further evidence of the significance of the memory effect.

6.3.6 *Load Imbalance*

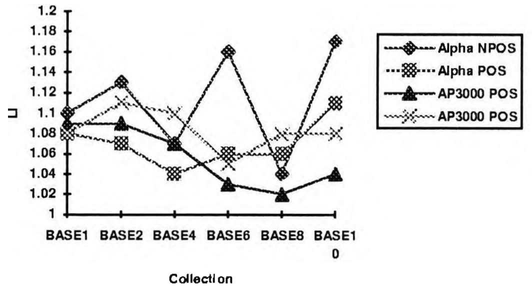


Fig 6-14. BASE1-BASE10 *local build* [*DocId*]: indexing load imbalance

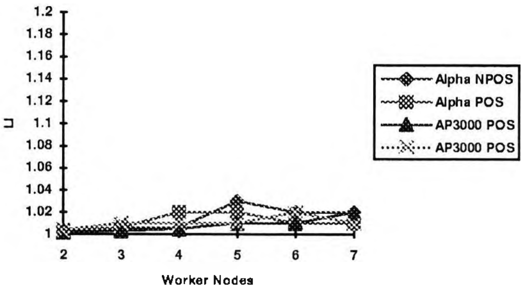


Fig 6-15. BASE1 *distributed build* [*DocId*]: indexing load imbalance

The load imbalance metric we use is declared in chapter 3, sub-section 3.4.1.8 - the ideal load balance is close to 1.0. In general it was found that the *distributed build* imbalance was lower than those of *local build* (see figs 6-14 and 6-15). In fact *distributed build* using any partitioning method is excellent on all nodes with both methods, e.g. on 2-7 Alpha and AP3000 *workers* the LI was in the range 1.002 to 1.03 on average for *DocId*. The LI figures demonstrate that the implemented process farm method provides good load balance for indexing jobs when whole files are distributed to workers.

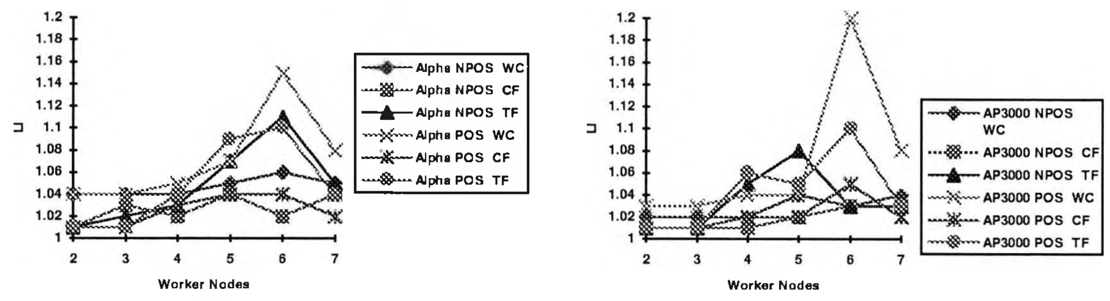


Fig 6-16. BASE1 *distributed build* [*TermId*]: indexing load imbalance

The results for *TermId* were generally not as good as *DocId*, but good in the average case (see fig 6-16). The exception was for builds with position data on 6 nodes: LI's of 1.2 for the AP3000 and 1.15 for the Alphas were recorded with word count (WC) distribution. The farm method described in chapter 4, section 4.4.2 is a very good way of ensuring load balance in the majority of cases. The *local build* LI is still very good: the worst LI recorded was 1.17 for BASE10 for the Alpha postings only run. We conclude by stating that both *Distributed* and *local build* methods achieve good load balance, but *local build* LI could be improved by paying more attention to text distribution.

6.3.7 *Merging Costs*

We consider here the percentage of time spent merging the temporary results to create the final inverted file: see chapter 3, section 3.4.1.10 for a formal definition - we look for the lowest possible cost in % terms. We examine the *DocId* method first. The merging for *local build* was in the main consistent within a 1% range, e.g. on the Alphas with posting data only, the average merge cost was 14 to 15% (see table 6-2). Merging costs for builds with position data were higher, e.g. on the Alphas the merge cost was 19 to 20%. Merge costs on the AP3000 were lower on *local build*, e.g. with posting data the average merge cost was around 13 to 14%. This difference is because the Alpha Farm processors are much faster and therefore the I/O time (which remains constant) is more significant.

With *distributed build DocId* build the merging costs were much the same as *local build* apart from Alpha builds with position data: the range found was 17 to 20%: these costs did not vary much from the *local build* (see table 6-3). The uniprocessor builds with position data registered the highest merge costs, whereas parallel *DocId* builds without position data saved indexes directly without the need for merging on 8 processors. The merge costs were more prominent on the Alpha as the faster processor speed reduces the computational costs and increases the importance of I/O (merge is an I/O intensive process). Merge costs are also more prominent on indexes which contain position data.

Collection	Alphas NPOS	Alphas POS	AP3000 NPOS	AP3000 POS
BASE1	-	20%	-	14%
BASE2	14%	19%	10%	14%
BASE4	14%	19%	9%	13%
BASE6	15%	19%	9%	13%
BASE8	14%	19%	9%	13%
BASE10	14%	19%	9%	14%

Table 6-2. BASE1-10 *local build* [*DocId*]: % of average elapsed indexing time spent merging

Work -ers	Alpha NPOS	Alpha POS	AP3000 NPOS	AP3000 POS
1	15%	24%	9%	16%
2	15%	20%	9%	14%
3	15%	19%	10%	14%
4	15%	20%	10%	14%
5	15%	19%	10%	13%
6	14%	18%	9%	13%
7	13%	17%	9%	14%

Table 6-3. BASE1 *distributed build* [*DocId*]: % of average elapsed indexing time spent merging

Work-ers	Alphas NPOS			Alphas POS			AP3000 NPOS			AP3000 POS		
Value	*WC*	*CF*	*TF*	*WC*	*CF*	*TF*	*WC*	*CF*	*TF*	*WC*	*CF*	*TF*
2	38%	37%	38%	44%	43%	44%	26%	26%	26%	35%	36%	37%
3	36%	35%	36%	42%	42%	42%	26%	26%	26%	34%	34%	34%
4	35%	34%	35%	41%	40%	40%	26%	26%	26%	34%	34%	41%
5	32%	31%	31%	36%	37%	36%	24%	25%	25%	31%	32%	38%
6	28%	29%	27%	33%	34%	33%	24%	24%	24%	29%	30%	30%
7	26%	26%	25%	30%	31%	31%	23%	23%	23%	28%	30%	29%

Table 6-4. BASE1 *distributed build* [*TermId*]: % of average elapsed indexing time spent merging: *distributed build*

Merge costs for *TermId* are very much higher as one would expect given the extra work required for merge with that method to exchange data between nodes (see table 6-4). These higher merge costs are a contributory factor in the overall loss of performance for *TermId* partitioning index builds. However there is a distinct decrease in all cases of the significance of merging on the Alphas, e.g. merging on indexes with position data and word count (WC) word distribution decreased from 44% at 2 *worker*s to 30% on 7 *worker*s. This is largely because the costs in transferring index data before the second merge can proceed increases with the numbers of worker nodes deployed, e.g. on the Alpha indexes with position data the increase is from 2 minutes at two *worker*s to 4 minutes at seven *worker*s. On the AP3000 a slight decrease in merging costs is recorded in most cases, and the decrease is not as pronounced as the Alphas. The Alpha's extra processor speed brings benefit to extra merging found when building *TermId* indexes. The corresponding figure for transferring indexes with position data on the AP3000 ranges from 2.4 minutes with two *worker*s to 2.9 with seven *workers*. The AP3000 is better able to cope with this extra cost in transferring data for the second merge as it has extra bandwidth available in its network.

6.3.8 *Summary of Time Costs for Indexing*

With respect to comparable metrics such as elapsed time and throughput, we have demonstrated that for a least one partitioning method, namely *DocId*, our results are state of the art compared with other VLC2 participants (Hawking et al, 1999). We have found that in most cases the Alpha farm outperforms the AP3000 except for some *TermId* runs: the AP3000 has a much higher bandwidth network available to it that is an advantage in such builds. Comparing the partitioning methods we have found that builds using the *DocId* method outperform index builds using *TermId* in all experiments. Our speedup and efficiency figures show that the methods of parallelism do bring time reduction benefits, particularly for the *DocId* partitioning method. The scalability and scaleup figures show that our implemented data structures and algorithms are well able to cope with increasingly larger databases on a same sized or larger parallel machine. The load imbalance is generally quite small for all runs. The extra costs for generating indexes with position data vary, but are not an insubstantial part of the overall costs. Merge costs are also an important element of total time, depending on the build and partitioning method used.

6.4 INDEX FILE SPACE COSTS AND PARTITIONING COMPARISON

In this section we declare the space overheads using the configurations described above. The results are compared and contrasted where necessary as well as comparing them with overheads on the BASE1 and BASE10 collections used in the VLC2 sub-track at TREC-7 (Hawking et al, 1999). The space overheads discussed are: overall inverted file space costs, keyword file space costs and file space imbalance.

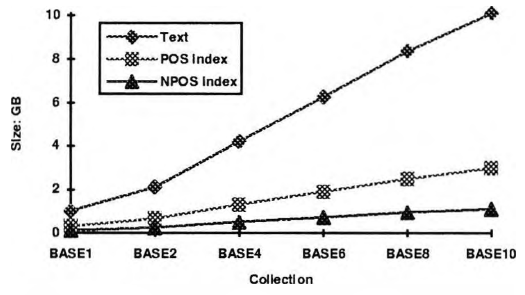


Fig 6-17a. BASE1-BASE10 *local build* [*DocId*]: index space costs in Gigabytes

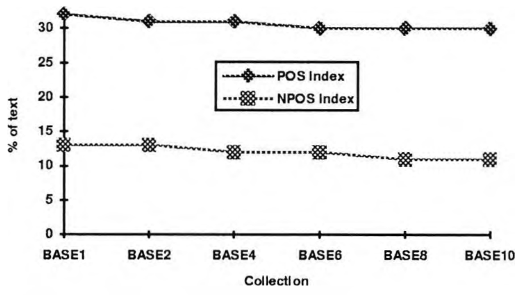


Fig 6-17b. BASE1-BASE10 *local build* [*DocId*]: index space costs in % of text

6.4.1 *Inverted File Space Costs*

The metrics we used here are the file sizes in Gigabytes and percentage of original text size. The space costs for *local build* indexes are fairly constant in percentage terms across all collections (see fig 6-17b), although a slight reduction in index size compared with the size of the text can be see in fig 6-17a. This reduction occurs irrespective of the type of data stored in the inverted file. From fig 6-18 we can observe that there is a slight increase in index size for increasing the processor set when using *distributed build* methods. The reason for this is because of the replicated map requirements of *distributed builds*. The increase is more marked for *DocId* partitioning. If the map file size is taken away from the total size then the *DocId* indexes increase is much smaller (the reason any increase at all is explained in section 6.4.2).

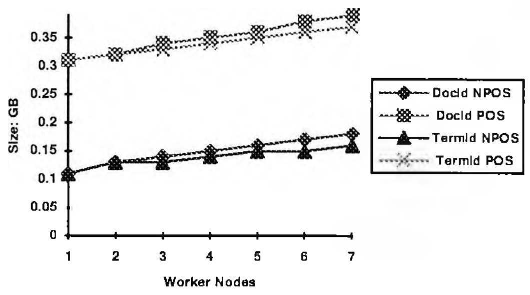


Fig 6-18a. BASE1 *distributed build*: index space costs in Gigabytes

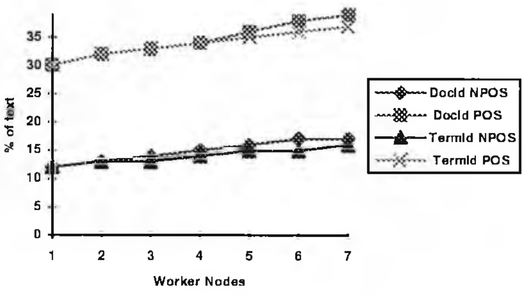


Fig 6-18b. BASE1 *distributed build*: index space costs in % of text

The comparison with space costs of the VLC2 participants [9] is favourable with postings only data: our smallest figure of 0.11 Gigabytes on BASE1 was smaller than all submitted results and on BASE10 only one VLC2 participant at 0.902 Gigabytes was smaller than our figure of 1.1 Gigabytes. The comparison with files that contain position data is not so good and our smallest figure of 0.31 Gigabytes for BASE1 is bested by two groups, while on BASE10 three groups record a smaller figure than our 3.0 Gigabytes.

6.4.2 *Keyword File Space Costs*

The metric for keyword file space costs is the size in megabytes and the keyword file percentage of the total inversion. With *local build* on both postings only and position data we found that the trend in keyword space costs was a decreasing one, e.g. 32% on BASE1 to 22% on BASE10 with postings only data (see fig 6-19b). This is because the increase in lexicon is not linear with the increase in collection (fig 6-19a). With *distributed DocId* indexes the keyword costs remain constant, e.g. 24-26% (see fig 6-20b). The size of the keyword file actually increases with more inverted file partitions (see fig 6-20a), but this increase is not significant and is absorbed by the increase in size of the replaced document map. We state that there is little extra cost in having words replicated across different fragments for *DocId* partitioning on this type of collection (Web data). For *TermId* indexes the size of the keyword file was constant irrespective of term allocation method, and if the map data is included in costs the significance of the keyword file with respect to the total index size gradually decreases (see figs 6-20a and 6-20b).

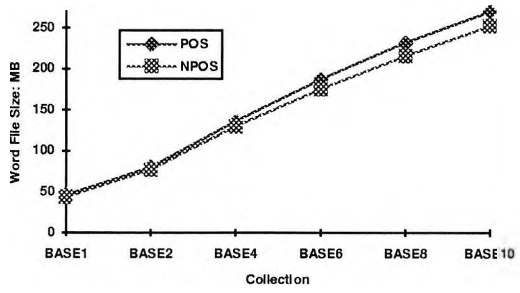


Fig 6-19a. BASE1-BASE10 *local build* [*DocId*]: index space costs in megabytes for keyword file

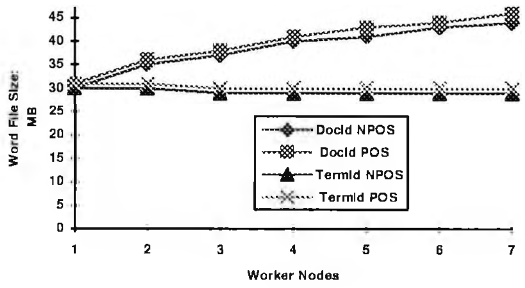


Fig 6-20a. BASE1 *Distributed Build*: space costs in megabytes for keyword file

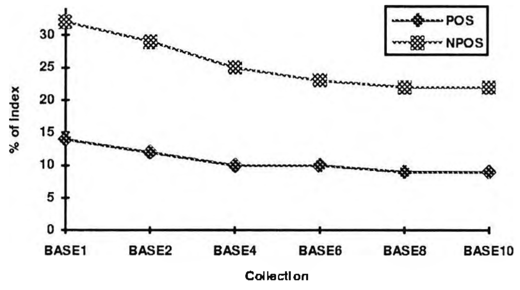


Fig 6-19b. BASE1-BASE10 *local build* [*DocId*]: index space costs in % of index for keyword file

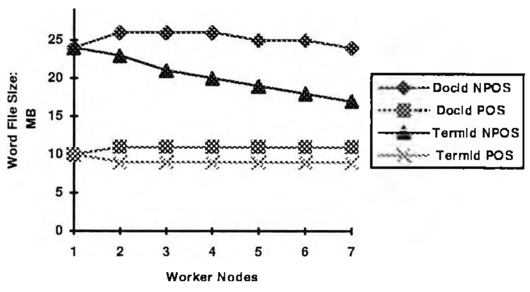


Fig 6-20b. BASE1 *Distributed Build*: index space costs in % of index for keyword file

6.4.3 *File Space Imbalance*

We use the concept of load imbalance (LI) but apply it to file sizes instead, i.e. maximum file size / average file size. We wish to ensure that index data is fairly distributed amongst nodes, e.g. it would not be desirable for one index partition to exceed the space available on a physical disk. The index time LI results are included in the figs 6-21 to 6-23 for comparative purposes. The space imbalance for text space costs was in general fairly stable being in the range 1.04 to 1.02 for all *local build* indexing runs (see fig 6-21). In comparison the inverted file imbalance was much higher, particularly for the smaller collections. Clearly the imbalance stems not from the size of the text, but from aspects of the text such as the number of documents and total word length of the text. In contrast the space imbalance for *distributed build* on *DocId* partitioning was small for any type of inverted file data storage (see fig 6-22). There is no significant difference between the space imbalance of inverted files and LI for indexing times with *DocId* partitioning. The file space imbalance figures further proof of the validity of the farming method for balancing load for *DocId* partitioning.

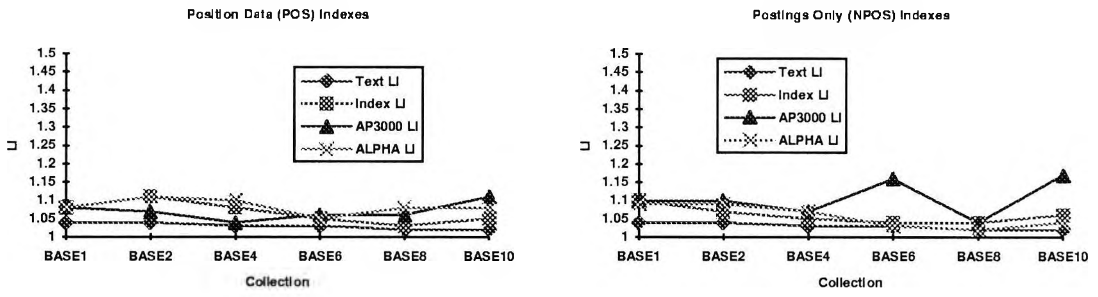


Fig 6-21. BASE1-BASE10 *local build* [*DocId*]: index space imbalance on files

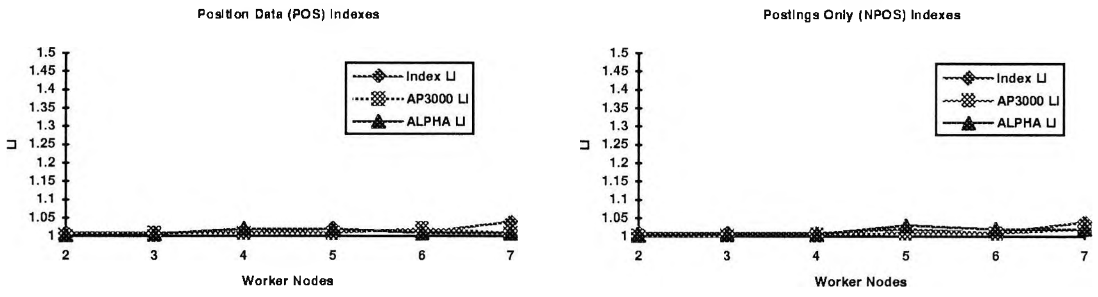


Fig 6-22. BASE1 *distributed build* [*DocId*]: index space imbalance on index files

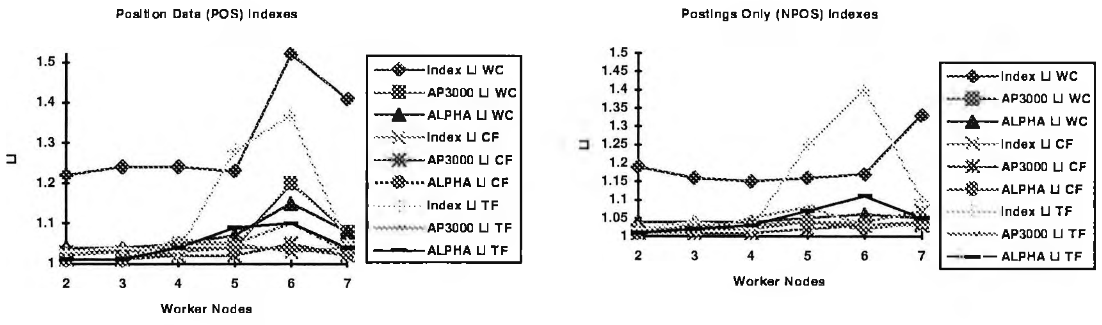


Fig 6-23. BASE1 *distributed build* [*TermId*]: index space imbalance on index files

The situation for *TermId* varies depending on the type of word distribution method used (see fig 6.23). For the word count (WC) distribution space imbalance was generally very poor, with the worst being indexes with position data on 6 worker nodes: an imbalance of 1.52 was recorded (interestingly the worst imbalance for indexing times, see fig 6.23). The figures for the collection frequency distribution method (CF) are much better with an imbalance range of 1.02 to 1.07 for all builds. In the term frequency (TF) method the imbalance was erratic being very poor at 5 and 6 worker nodes for any index builds, but good on all other runs. Any imbalance in space does not affect computational imbalance adversely. None of the *TermId* space imbalance results are as good as the *DocId* for space costs on *distributed builds*, as it is much harder to derive a good data distribution method for *TermId* indexes (the allocation of terms to nodes is a more difficult problem than allocating documents to nodes). None of the methods implemented affect space imbalance such that an index partition exceeds the physical disk of any node.

6.4.4 *Summary of Space Costs for Indexing*

Overall space overhead for the indexing is state of the art and comparable with the results give by VLC2 participants: at least for indexes with postings only. The *Distributed Build DocId* results show that the cost of storing keywords does grow with increasing the fragmentation, but given that *local build* results show that space costs decrease with database size we do not see this a serious overhead for the *DocId* partitioning method. The space costs imbalance for *local build* is generally quite stable, but the generated inverted files vary more. Clearly the consideration of the number of files on its own is not sufficient to ensure very good balance. For *distributed builds* space imbalance was much smaller, except for some *TermId* indexes where distribution methods are more difficult to derive: no index partition exceeds the size of a node's local disk.

6.5 TREC-8 EXPERIMENTS

Given that experiments on the *DocId* method proved to be superior than *TermId* in the experiments above, we decided to use the partitioning method in conjunction with *local builds* in our TREC8 experiments (MacFarlane et al, 2000a). A total of 17 processors was used in the "Cambridge Cluster" to map 16 Indexer processes and 1 timer process. We indexed the TREC8 Ad-Hoc collection together with the full 100 Gb VLC2/WT100g collection and its baselines (BASE1 and BASE10). We also completed an indexing job on the Ad-Hoc data using a single Pentium processor for comparison purposes. All the collections under investigation were distributed as evenly as possible across the 16 nodes of the Cluster by linear assignment i.e. 1st x files are placed on the 1st node, 2nd x files are placed on the 2nd node etc: x is approximately total collection files divided by the number of nodes.

6.5.1 *Ad-Hoc Track Experiments*

Table 6-5 show details of Ad-Hoc Indexing experiments. Indexes with and without position data were produced. The cluster yielded good results particularly on Indexes with position data where the extra memory on the "Cambridge Cluster" paid dividends: super-linear speedup and efficiency were recorded. There is a slight increase in index size on the "Cambridge Cluster" due to repetition of keyword records found in the type of Index used. The increases are only minor however: 0.05% for postings only files and 0.02% for files with position data.

Inverted File Type	**Machine**	**Time (Hours)**	**Speedup**	**Efficiency**	**LI**	**Index Size (% of Text)**
NPOS	Pentium	0.81	-	-	-	324 Mb (17%)
	Cluster	0.059	13.68	0.85	1.06	342 Mb (18%)
POS	Pentium	1.04	-	-	-	832 Mb (43%)
	Cluster	0.064	16.17	1.01	1.06	851 Mb (47%)

Table 6-5. TREC8 Ad-Hoc indexing experiment details

6.5.2 *Web Track Experiments*

COLLECTION	WT100g	BASE10	BASE1
Index Time (hrs)	3.04	0.29	0.025
LI	1.10	1.06	1.10
Scalability	BS10:0.91 BS1: 0.8	BASE1:0.87	-
Index Size in GB (% of Text)	10.64 (11%)	1.21 (12%)	147MB (14%)
Collection Word Length (Million)	8,600	865	87

Table 6-6. Web track Indexing experiment details

Details of our Web Track indexing experiments are given in table 6-6. The indexing times for the Web Track collections compare favourably with the results given the 1999 VLC2 track: the times stated above are faster than all VLC2 indexing times and meet the standard sought at VLC2 (an indexing time of 10 hours or less). They are also the best figures for the Web Track (Hawking et al, 2000). The load balancing for all Indexing experiments on the Web Track is good with only slight levels of imbalance recorded: this confirms that the strategy used for distributing the collection to nodes was a good one. The index time scalability from the baselines to WT100g and from BASE1 to BASE10 are good, with very little deterioration in time per index unit. The index sizes also compare very well with other VLC2 or Web Track participants: only two groups yielded smaller indexes than the figures we quote in table 6-6. The indexes produced on all collections contained postings only data.

6.6 CONCLUSION

The results produced in this chapter show that of the partitioning methods, *DocId* partitioning using any build has by far the most promise (as predicted by theory in chapter 5, section 5.2.2) and would in most circumstances be the method chosen for indexing. This would be the case particularly if the collection under consideration needed frequent re-builds. We have used the *DocId* method to good effect in the Web track for TREC-8 on the full 100 Gigabyte VLC2 collection. Where disk space was limited, the *local build* method could be used to good effect: we used this build method on the BASE10 as we did not have sufficient space to do *distributed builds* on that collection. We have demonstrated that indexing is state of the art in both compute and space terms by comparing our space and time results with those given at VLC2 (Hawking et al, 1999) and the TREC-8 Web Track (Hawking et al, 2000). Although we did not produce the best results for all measures, no group at VLC2 did either. Our indexing time for the full 100Gb collection was the best in the Web Track. When comparing the performance of both partitioning methods, our empirical results show the synthetic model for indexing is able to predict that *DocId* builds are faster than *TermId* builds. However, problems in the modelling of communication in the model meant that a widening gap in performance between the two partitioning methods for increasing parallel machine size was not anticipated in the model.

A clear distinction must be made between *DocId* and *TermId* partitioning methods. *Distributed build DocId* out-performs *TermId* in all areas of time cost metrics and would therefore always be preferred if indexing was of primary concern. We state this irrespective of the type of inversion or algorithms/methods used if cluster computing is utilised. This allows us to make generic statements on performance of indexing and further strengthen our assertions made in the synthetic model. We would recommend that *TermId* only be used if two main criteria are met. One is that a high performance network is available to reduce time spent on transferring data during the *global merge* process. The other is that some other benefit must accrue from the use of *TermId* partitioning which in essence would be some advantage in search performance or index maintenance criterion over the *DocId* method.

Chapter 7

Probabilistic Search Results

7.1 INTRODUCTION

In this chapter we describe experiments and results for the probabilistic search task aspect of our thesis. Our main aim is to compare search on inverted files given two types of partitioning methods under consideration. Most of the material in this chapter was published in MacFarlane et al, (2000b). The experimental aims and objectives of our research are given in section 7.2. The data and settings used for the experiments described in 7.4 and 7.5 are described in section 7.3. The results of searches on the two chosen partitioning methods are reported in sections 7.4 and 7.5. Section 7.6 compares and contrasts these results with each other and our stated aims/objectives. We then describe our TREC8 web track experiments (MacFarlane et al, 2000a) in section 7.7 using the best performing partitioning method discovered in previous experiments. A summary and conclusion is given in section 7.8.

7.2. EXPERIMENTAL AIMS AND OBJECTIVES

Earlier work on the subject of search performance on partitioned inverted files [2,4] used simulations. We use real Web collections and apply TREC queries to examine performance. The query model used in these simulations either assumes an equal probability of words occurring in a query (Tomasic and Garcia-Molina, 1993a) or uses both term skew and uniform term distribution models (Jeong and Omiecinski, 1995). We believe that what the user does has a significant impact on the performance of a parallel system using the partitioning methods discussed. We put forward a hypothesis that focuses on query size: since users tend to submit smaller queries (Kirsch, 1998; Silverstein et al, 1999), only one or two of the nodes in the system will be servicing a query using *TermId.* Therefore for a given query the load balance will always be skewed as many nodes will be doing no work at all: this skew may accumulate with increasing numbers of queries. We define load balance as the spread of a given computation across a number of nodes in a parallel computer: ideally all nodes should have the same computational load. In the *DocId* method however, each node is involved in the processing of all terms and therefore load balance for a single query is likely to be much superior to *TermId.* Our hypothesis is therefore;

> The *DocId* partitioning method will perform better on probabilistic search because 1) users tend to submit short queries, which points to a load balancing

problem for the *TermId* method, and 2) if documents are evenly distributed, *DocId* will produce good load balance as all nodes in the parallel machine process the query.

Our aim is to test the claims in the hypothesis as they relate to sequential query service. We do not address the issue of dynamically migrating indexes or concurrent query service. It should be noted that we principally address the issue of retrieval efficiency in this study, but mention retrieval effectiveness in order to verify that searching *TermId* and *DocId* partitioned indexes yield the same effectiveness results.

7.3. DATA AND SETTINGS USED

The data used in the experiments comprised the BASE1 and BASE10 sub-sets of the official 100 Gigabyte VLC2 collection (see chapter 3, section 3.3.1). We applied search to indexes with and without position data. We use two types of builds for indexes: *distributed builds* and *local builds*. For the *distributed build* method we use the BASE1 collection only, creating indexes on 1 to 7 nodes using both types of partitioning method and initiating searches on all of those indexes. The BASE1 and BASE10 collections were used for the *local build* method, running queries on 8 nodes. We use different process to processor mapping strategies for each type of build. In *distributed build* we map the client and top node to a single processor, separate from each of the leaf nodes. For *local build* we have to map the client and top node to one of the leaf nodes because of restrictions on the available nodes of the AP3000 partition set.

Collection	**Query Type**	**prec. @ 5**	**prec. @10**	**prec. @15**	**prec. @20**	**K1**	**B**
BASE1	*title only*	0.244	0.178	0.149	0.130	1.5	0.2
	whole topic	0.188	0.172	0.145	0.128	1.5	0.4
BASE10	*title only*	0.324	0.282	0.273	0.264	1.5	0.2
	whole topic	0.356	0.298	0.271	0.247	1.4	0.7

Table 7-1. Retrieval effectiveness results for main search experiments

The queries are based on topics 351 to 400 of the TREC-7 ad-hoc track: 50 queries in all (Hawking et al, 1999). The terms were extracted from TREC-7 topic descriptions using an Okapi query generator utility to produce the final queries. We used two types of queries: one based on *title only* (average number of terms per query is 2.46) and one based on the *whole topic* (average number of terms per query is 19.58). The *whole topic* query set has 51 queries, one extra being for VLC2 experiment initialisation (Hawking et al, 1999). Our experiments concentrated on *title only* queries as users have a tendency to issue smaller queries. Table 7-1 shows retrieval effectiveness results on these queries together with the BM25 tuning constants used. Retrieval effectiveness results for a given query type were identical on all runs irrespective of inverted file type used, partitioning method applied or number of nodes used. It should be noted that we were unable to get the same level of retrieval effectiveness on longer queries as with shorter queries: this merits further investigation.

Query Type	**File Type**	**Time (Secs)**
title only	NPOS	0.110
	POS	0.216
whole topic	NPOS	2.45
	POS	6.02

Table 7-2. BASE1: uniprocessor average elapsed times in seconds

Our timing methodology was as follows: we declare the average of 10 runs. We declare results from the AP3000, but did runs on the Alpha farm to check portability of the code using evaluations to check that precision was identical. Sequential run times for comparison with parallel runs are declared in Table 7-2: these times are well within the 10 second criteria for elapsed time suggested by Frakes (1992). Sequential runs were done on one node with a single search process. Note that runs with postings only data is signified by NPOS file type, while runs on indexes with position data are signified by the POS file type. Although we do not use position data in our weighting functions, we need to examine the effect of position data on probabilistic search performance as in a deployed system users may submit queries with adjacency operations or may request passage retrieval. We assume that the position information is stored contiguously with each posting.

For the *TermId* method three strategies for term allocation to partitions are used. The most basic of these is the WC method that allocates terms to partitions on the basis of word count in 100 buckets. A heuristic is used to allocate terms as equally as possible on this basis. The other two methods allocate on collection frequency (CF) and term frequency (TF) basis. More details on these term allocation methods can be found in chapter 4, section 4.4.2.

We use a number of metrics to measure the performance of our program: average query processing time in seconds, speedup, efficiency, load imbalance (LI), extra cost ratio in time of inverted files containing position data, overheads such as compute and wait time in the top node and scalability (these metrics and our requirements of them are declared in chapter 3, section 3.4.1).

7.4 SEARCH RESULTS FROM DOCID PARTITIONING

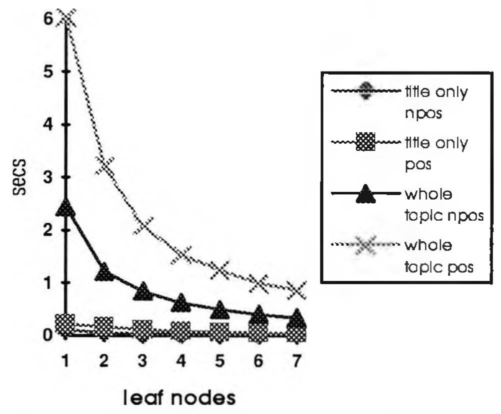


Fig 7-1. BASE1 [*DocId*]: search average elapsed time in seconds

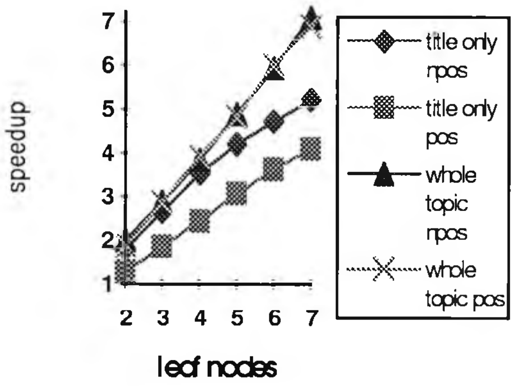


Fig 7-3. BASE1 [*DocId*]: search speedup

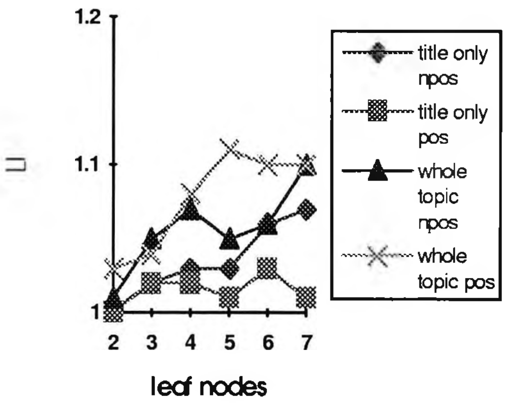


Fig 7-2. BASE1 [*DocId*]: search load imbalance

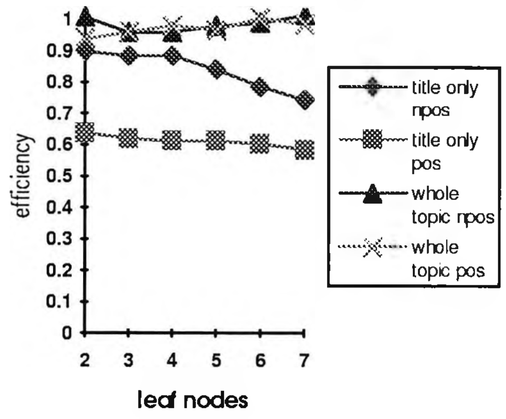


Fig 7-4. BASE1 [*DocId*]: search parallel efficiency

The results for this type of partitioning are encouraging (see figs 7-1 to 7-4). For *title only* queries the best elapsed times are very good indeed (see fig 7-1). Response times of 21 milliseconds are recorded for inversion with postings only data and 53 milliseconds for position data indexes. The comparison with VLC2 average response time is favourable: our *title only* time matches the best VLC2 results (Hawking et al, 1999) at 2 leaf nodes and betters it on 3 to 7 leaf nodes. For *whole topic* the best elapsed times are a third of a second for postings only and 0.87 seconds for files with positions. All runs (including BASE10 runs) therefore meet the 10 second criterion for search times suggested by Frakes (1992). Our BASE10 results are only bettered by one participant of VLC2 (Hawking et al, 1999). Time reduction relative to uniprocessor runs is exhibited by all multiprocessor runs.

The result from these times is that good to reasonable speedup can be found on most parallel runs, being very near linear for *whole topic* and *title only* with postings only (see fig 7-3). Speedup on *title only* queries serviced on files with position data is disappointing however: the extra I/O needed to retrieve sets from disks outweighs any gain through parallelism on the sort for smaller queries. The efficiency results reflect the speedup figures (see fig 7-4) and are particularly good for *whole topic* queries, results on all runs with this query type are very near 1. Load imbalance is not an issue as for any query the load imbalance (LI) was found to be just over 1 in all cases (see fig 7-2). Imbalance was more noticeable on *whole topic* queries than *title only*, however due to the difference in set sizes between nodes.

Leaf nodes	***title only***		**Whole Topic**	
-	NPOS	POS	NPOS	POS
2	2.0%	1.1%	0.0%	0.0%
3	4.7%	1.3%	0.0%	15.5%
4	3.8%	2.8%	2.1%	0.07%
5	11%	4.3%	0.0%	0.0%
6	15%	4.8%	2.0%	1.1%
7	17%	8.2%	5.2%	1.2%

Table 7-3. BASE1 [*DocId*]: search overheads in %

Data	***title only***		**Whole Topic**	
-	NPOS	POS	NPOS	POS
BASE1 time (secs)	0.027	0.057	0.32	0.77
BASE10 time (secs)	0.18	0.54	4.18	6.45
Scalability	1.48	1.06	0.826	1.19

Table 7-4. BASE1/BASE10 [*DocId*]: search scalability

Overheads on the top node were found to be very small for *title only*, however there is a trend upwards which is particularly noticeable on runs using postings only indexes (see table 7-3). For *whole topic* they were insignificant with other aspects of the search dominating time: in some runs overheads were too small to be recorded. The extra cost on search time for having position data in inversion was found to be constant for *whole topic*: search was around 2.5 times longer than for postings only data. For *title only* the extra costs were worse on multiprocessor runs than uniprocessor runs and varied a great deal more than *whole topic*. This factor is further evidence of the extra burden I/O places on smaller queries with posting list containing position data. The scalability of the parallel program on 8 leaf nodes is good with very good figures for *title only* queries (scalability is over 1) and for *whole topic* (see table 7-4).

7.5. SEARCH RESULTS FROM TERMID PARTITIONING

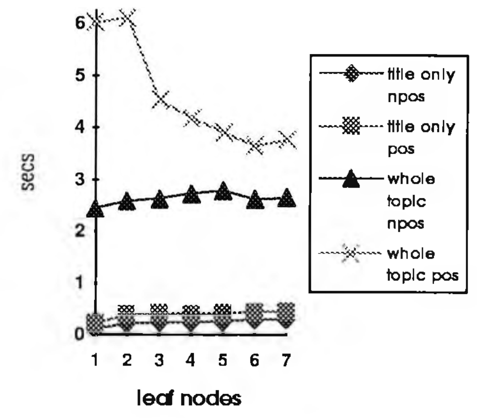


Fig 7-5. BASE1 [*TermId*]: search average elapsed time (sequential sort: WC distribution)

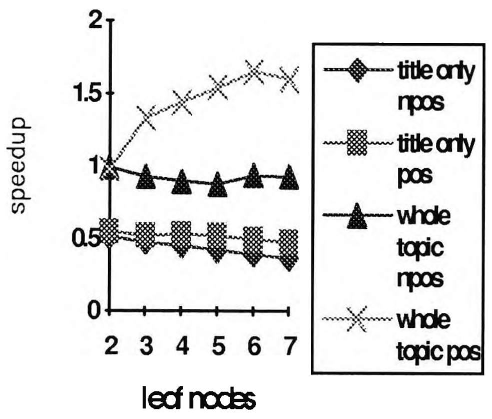


Fig 7-7. BASE1 [*TermId*]: search speedup (sequential sort: WC distribution)

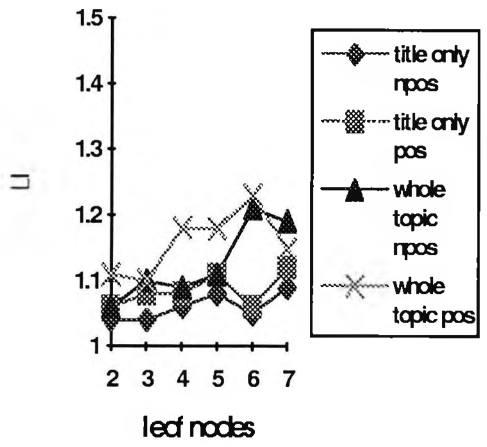


Fig 7-6. BASE1 [*TermId*]: search load imbalance (sequential sort: WC distribution)

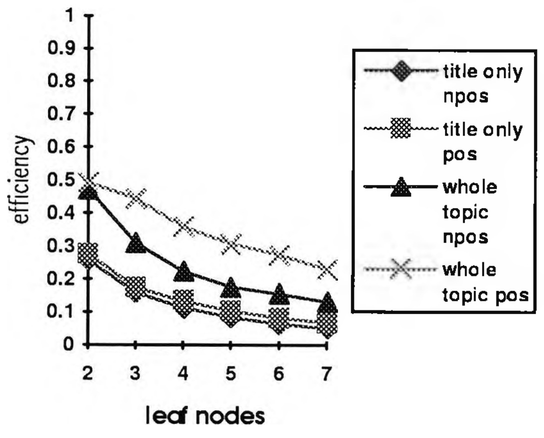


Fig 7-8. BASE1 [*TermId*]: search parallel efficiency (sequential sort: WC distribution)

The results with this type of partitioning are discouraging (figs 7-5 to 7-8 show results for using indexes with word count (WC) distribution: full details of other retrieval efficiency results can be found in appendix A2). While all the runs on any type of query meet the 10 second criteria for search times, there is no advantage from parallelism with this type of partitioning method (see fig 7-5). Times do not compare favourably with VLC2 participant times (Hawking et al, 1999). The only example of an elapsed time decrease is *whole topic* queries serviced on indexes with position data, therefore slowdown is recorded in most cases (see fig 7-7). The effect on efficiency is dramatic with figures ranging from poor to unacceptable: a linear reduction in efficiency is recorded in all cases (see fig 7-8). The implication of this is that scalability would be poor: we therefore did not do any experiments on the BASE10 collection. Load imbalance is generally very small for *title only* queries, but is perceptibly worse for *whole topic* queries: we cannot guarantee that terms will be fairly distributed to nodes with larger queries. It should be noted that the load imbalance (LI) figures on *title only* are averaged over 50 queries on 10 runs so the evidence available suggests that computational skew does not accumulate with increasing numbers of queries (see fig 7-6). The overheads at the top node are clearly a problem with this partitioning and topology scheme, and are much higher than for *DocId* partitioning (see table 7-5): a big factor being the sort component of probabilistic search. The overheads for both *title only* and *whole topic* queries take up more than 50% of the total search time: the overheads are less pronounced on indexes which contain position data, but are still significant. For costs of position data on probabilistic search it was found that overheads tend to reduce with increasing number of leaf nodes, a factor more marked in *title only* queries. The reason for this is that runs on indexes with position data gain more from I/O parallelism than runs on indexes with postings only, as indexes with positions contain much more data.

Leaf nodes	*title only*		*whole topic*	
-	NPOS	POS	NPOS	POS
2	63%	40%	53%	26%
3	68%	46%	62%	40%
4	72%	52%	69%	46%
5	75%	54%	75%	50%
6	78%	59%	76%	54%
7	79%	59%	75%	49%

Table 7-5. BASE1 [*TermId*]: search overheads in % (sequential sort: WC distribution)

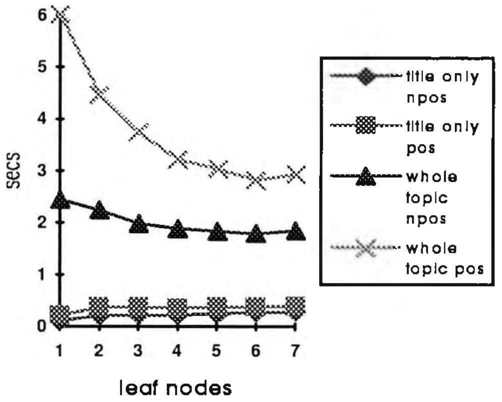


Fig 7-9. BASE1 [*TermId*]: search average elapsed time (parallel sort)

Having identified a major bottleneck for the sort when using *TermId* search (initially found in the synthetic model - see chapter 5, section 5.3.2), we thought it worth while having an attempt at speeding up the sort by using a parallel method. This required the generation of the final set as normal, but elements of the sort were scattered to leaf nodes that applied a sort to their section of the final result set. The top ranked 20 documents were then gathered as per

DocId method. The results however did not improve the performance of the parallel program much (see figs 7-9 to 7-12 and table 7-6). Average query processing times were reduced slightly (see fig 7-9), but speedup/efficiency for *title only* queries were still poor, while *whole topic* queries did show some slight speedup gains (efficiency was still poor) - see figs 7-10 and 7-11. Load imbalance (LI) remained small on all runs (see fig 7-12). The basic problem with this revised method is that in applying the parallel sort, the amount of communication overhead is increased such that most or all of the gain made in the parallel sort is lost.

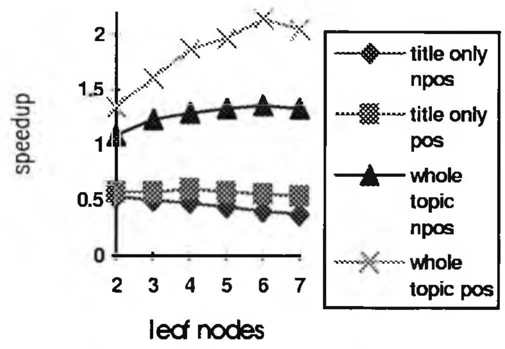


Fig 7-10. BASE1 [*TermId*]: search speedup (parallel sort)

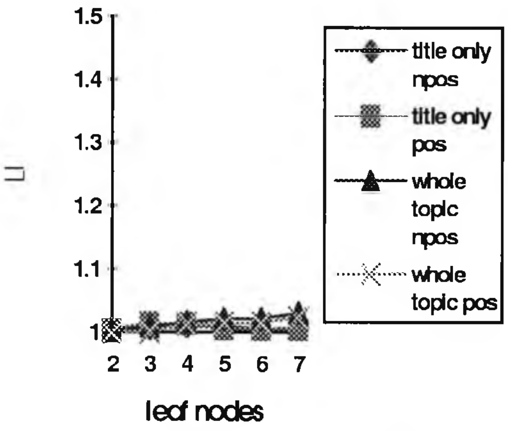


Fig 7-12. BASE1 [*TermId*]: search load imbalance (parallel sort)

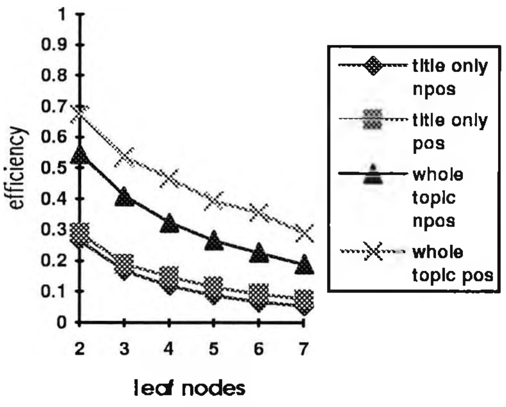


Fig 7-11. BASE1 [*TermId*]: search parallel efficiency (parallel sort)

Leaf nodes	*title only*		*whole topic*	
-	NPOS	POS	NPOS	POS
2	3.0%	0.6%	0.2%	0.0%
3	5.1%	2.0%	1.5%	0.4%
4	5.6%	2.4%	2.3%	0.2%
5	5.5%	3.5%	3.1%	0.5%
6	6.1%	3.8%	4.5%	1.0%
7	5.6%	3.9%	3.5%	0.7%

Table 7-6. BASE1 [*TermId*]: search overheads in % (parallel sort)

7.6 COMPARISON OF PARTITIONING METHODS

In the context of the experiments defined in this chapter, *DocId* partitioning is clearly the preferable method for search (for a different perspective on the comparison between methods see figs 7-13 and 7-14 which use the throughput metric). While both methods show acceptable response times, only *DocId* shows any real benefit when multiprocessors are used. The impact on throughput can be seen in figs 7-13 and 7-14 where *DocId* records linear increase for both types of indexes, while *TermId* partitioning results are disappointing. *TermId* partitioning clearly fails on parallel measures such as speedup and efficiency. The problem with *TermId* is that too much data has to be communicated from the leaf node to the top node and the sort cannot be parallelized without further communication between leaf nodes and the top node. The reason for the communication overhead with *TermId* is that a final result for any given document cannot be guaranteed to be completed in a single leaf node. The overhead at the top node is a serious bottleneck with *TermId* and cancels out any advantages in parallelism applied to I/O, set weighting merging and sorting. Unless some other topology could be found which would support the cluster of workstations model we use, we do not see *TermId* partitioning as a viable technique for parallelizing probabilistic search. In order to support *TermId* we could perhaps use a more complicated topology. We do not see the point in attempting this when we have a partitioning method that works well on a simple topology. It could be argued that another type of architecture may be well suited to the *TermId* method. It is difficult to sustain such an argument given the dominance of the cluster of workstations model in parallelism today. The results presented here confirm that *TermId* partitioning for distributed inverted files is not viable in our parallel search context.

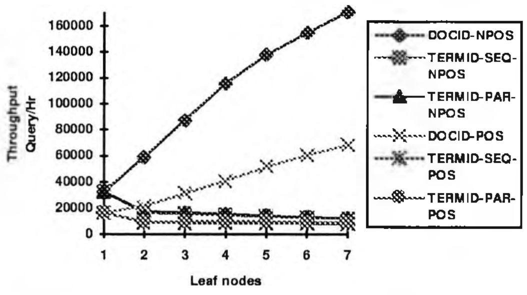


Fig 7-13. BASE1: throughput for *title only* query set

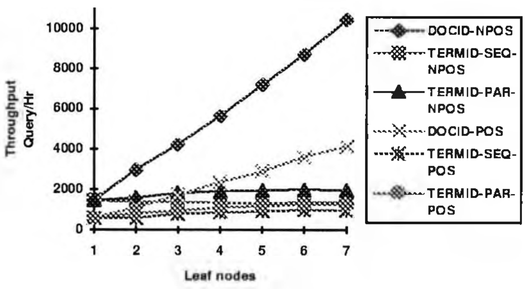


Fig 7-14. BASE1: throughput for *whole topic* query set

How do the empirical results relate to the hypothesis stated in section 7.2; i.e. does the evidence refute or confirm the hypothesis? The second part of the hypothesis that suggesting good load balance searching on inverted files using the *DocId* partitioning method, is confirmed. The first part on the *TermId* method needs to be revised however as it has been demonstrated that good load balance can be achieved as computational skew does not accumulate. The evidence applies to small queries only where the times are averaged over 50 queries. The imbalance for larger queries is much worse, but since users tend to submit smaller queries we cannot use that line of argument to defend the hypothesis as it stands. The good average load imbalance (LI) on *title only* queries are found irrespective of the term allocation to inverted file partition mechanism used. It is therefore possible that *TermId* may be able to offer a better concurrent query service than *DocId* particularly with respect to throughput. It should be stated however that although part of the hypothesis has been confirmed, technological factors are more important than information theoretic ones in probabilistic search. The sort needed for ranking is a considerable cost (O(n*log*(n) for Quicksort), and *TermId* cannot apply parallelism without increasing communication costs, a disadvantage the *DocId* method does not suffer from.

A number of different assertions have been made with respect to partitioning search and we examine them given the evidence provided above. Jeong and Omiecinski (1995) concluded that if the terms are less skewed in distribution *TermId* would be the best method to choose, while a skewed term distribution in queries would suggest that *DocId* would be a better partitioning method. Given that users tend to submit smaller queries it seems axiomatic that the *DocId* method would be preferable. However the evidence produced above suggests other aspects of term weighting search are more important than the distribution of terms in the query, in particular the sort to produce the final ranked set. Search time on *DocId* partitioning indexes is superior in this aspect. It should be stated that our results validate the assertion that searches on *DocId* partitioning indexes is better when term distribution is skewed. Tomasic and Garcia-Molina's (1993a) results indicate that a hybrid of the partitioning methods discussed in this chapter yields better response times (see chapter 2, section 2.4.4 for a discussion on these methods). While their results are not directly comparable with ours we can make assertions on a higher abstract level. They use a slightly different architectural model in which each node has two disks attached to a cluster node rather than one in our model (the architecture can still be classed under shared nothing, however). Their simulation suggests that in the best performing strategies the data for one document was kept on one node, and our results validate theirs. Local sorts can be applied using their strategies and the same reduced communication load can be obtained.

7.7 TREC-8 LARGE WEB TRACK EXPERIMENTS

The purpose of our Web Track experiments was to examine the scalability of the implemented data structures and algorithms as well as contribute to the debate on centralized versus distributed web search indexes. We submitted three runs, one for the full WT100g and the other two for the baselines. We ran all 10,000 of the chosen web queries against each of the databases, using the same query configuration for each run using 18 processors of the Cambridge Cluster. We used *local build* indexes with postings only data in the inverted list. We used term weighting search with the BM25 weighting function. Details of the Web Track runs are given in table 7-7. We also report details of our preparatory VLC2 experiments. This material was first published in MacFarlane et al (2000a).

TRACK RUN-ID	QP TYPE	DATABASE	COMMENTS
plt8wt1	Term W.	WT100g	1 Timing run
plt8wt2	Term W.	BASE10	1 Timing run
plt8wt3	Term W.	BASE1	1 Timing run

Table 7-7. Details of web track search runs

7.7.1 *Retrieval Effectiveness*

VLC2 Experiments

QUERY GEN.	BASE1	BASE10	VLC2/ WT100G	QUERY SIZE
Title	0.102 (0.130)	0.235 (0.264)	0.318 (0.377)	2.46
Title/Descr.	0.117	0.256	0.370	9.46
Title/Descr/Narr	0.103	0.228	0.392	26.54
Okapi VLC2	0.111	0.240	0.429	19.34

Table 7-8. Precision at 20 for VLC2 experiments

We present the retrieval effectiveness results on the VLC2 data in table 7-8. The figures in brackets are evaluations with tuning constants set as: **K1**= 1.5 and **B** = 0.2. These were found to be the best combination on VLC2 data evaluations when doing our tuning constant variation experiments. The other runs are with tuning constants set as **K1**=2.0 and **B** = 0.6: these settings were used in our original VLC2 experiments at TREC-7 (MacFarlane et al, 1999b). We declare runs on generated queries based on the Title, Description and Narrative as well as the Okapi generated VLC2 queries we used in TREC-7. Queries were generated from topics 351 to 400.

We have managed to improve the retrieval effectiveness our system considerably over our VLC2 entry (MacFarlane et al, 1999b). For example comparing the results for our original VLC2 entry using Okapi VLC2 queries, we have a figure of 0.111 compared with 0.08 and 0.056: an improvement of 39% and 98% respectively. Examining the tuning constant data allowed us to improve our *title only* queries quite considerably (we did not do experiments on other query types because of time constraints). Our *title only* results compare favourably with TREC-7 VLC2 runs where 0.377 is higher than 4 out of 18 submitted runs (which were based on tile/description in the main). However our Okapi VLC2 query run results are only higher than 5 out of the 18 submitted runs and we therefore need to examine the issue of effectiveness on larger queries (including experiments with tuning constant variation). As with other participants in the TREC-7 VLC2 track, we also recorded a significant rise in precision at 20 moving from the baselines to the full collection (Hawking et al, 1999).

Web Track Experiments

The results from the 50 evaluated Web Track queries are very good indeed on all collections, particularly the full 100 Gbyte collection (see table 7-9). The most important aspect of these experiments is that the tuning constants chosen in *title only* VLC2 runs were good predictors for retrieval effectiveness for Precision at 20: the chosen **K1** and **B** values were best in the 50 evaluated Web Track runs. The trend in both tuning constant data sets is very much the same. We can clearly state from the evidence provided with the data used in the Web track experiments that the hypothesis stated in chapter 3, section 3.2.1, holds when the source and target collection are the same. This demonstrates the possibility of examining statistics from a given collection (and perhaps using some form of heuristic process) in order to choose a given pair of tuning constants for incoming queries. Only one group participating in the 2000 web track registered a better retrieval effectiveness on the full collection than our runs.

DATABASE	MODIFIED AV PREC	PREC @ 10	PREC @ 20
BASE1	0.189	0.320	0.269
BASE10	0.323	0.476	0.436
WT100g	0.458	0.550	0.561

Table 7-9. Large web track retrieval effectiveness results

7.7.2 *Retrieval Efficiency*

VLC2 Experiments

QUERY GEN.	BASE1	BASE10	WT100g
Title	0.052	0.074 (1.4)	0.87 (16.8, 11.78)
Title/Descr.	0.063	0.339 (5.4)	4.4 (69.8, 12.99)
Title/Descr/Narr	0.11	1.06 (10.05)	14.64 (138.2, 13.76)
Okapi VLC2	0.14	0.93 (6.83)	12.21 (89.9, 13.2)

Table 7-10. Average query processing times in seconds for VLC2 experiments (ratios to baselines)

In table 7-10 we present the average query processing times in seconds together with the ratios to the appropriate baseline measures. The ratio is defined as: Big Collection Response time/Little Collection Response Time (Hawking et al, 1999). We give two figures for scalability in our results for comparison purpose; one relating to the actual text size and one to the Inverted file size. The processing speed for most runs is very good. We report the average of the two runs submitted for each database. Only two of the VLC2 runs exceed the 10 second requirement for query response times, but these are the two largest queries applied to the index (see table 7-8 for query sizes). One of the runs, namely *title only* meets the VLC2 requirement of a 2 second or less response time for queries over the full collection (Hawking et al, 1999) and compares favorably to query response times for VLC2 participants. Query processing times in proportion to the collection size showed sub-linear growth when comparing BASE1 to the other collection runs: the exception was the larger Title/Description/Narrative runs. The ratio from BASE10 to WT100g shows super-linear growth for elapsed times for all runs.

The Scalability results are given in table 7-11. The best scalability results are found with *title only* queries, some of which are very spectacular for both text and index Scalability. It is clear from the data that there is a strong correlation between query size and scalability: as we increase the size of the query, scalability declines. We have organised table 7-11 in descending order of Scalability so this can be clearly seen. This effect is due to memory use: more constituent query terms mean that larger numbers of sets have to be manipulated: this effect would clearly be more of a problem on uni-processor experiments. Some level of query processing optimisation may therefore be required for larger queries (Hawking, 1998).

QUERY GEN.	**BS1-BS10**		**BS1-100g**		**BS10-100g**	
	INV	**TXT**	**INV**	**TXT**	**INV**	**TXT**
Title	5.953	7.03	4.43	5.76	0.744	0.819
Title/Descr.	1.581	1.867	1.07	1.39	0.675	0.743
Okapi VLC2	1.239	1.464	0.826	1.07	0.670	0.733
Title/Descr/Narr	0.843	0.996	0.537	0.698	0.637	0.701

Table 7-11. Scalability results for VLC2 search experiments

Web Track Experiments

Retrieval efficiency results for the web track experiments are presented in table 7-12. The processing speed for all runs is also very good, but not as good as for *title only* VLC2 runs. However, all runs meet the 10 second requirement for query response times. Load balance is good for all runs. The prepared average query length was found to be 2.49 terms: many terms in the queries were non-content bearing words and some queries contained none at all. A frequent query was the single term "a". Why was the response time for Web Track queries higher than *title only* on WT100g (nearly double) despite the fact that Web Track queries are only slightly larger than *title only* queries? We compared the average set size of the 50 *title only* queries used in our VLC2 experiments to a sample of 50 Web Track queries. We found that the average set size per term for *title only* queries was 28K, whereas for the sample Web Track queries the average set size was 48K. The memory requirements for sets are very much larger for Web Track queries than for the *title only* set. We experimented with differing levels of in-core keyword on the sub-set of Web Track queries and found that we got best results with only 9% of the keyword dictionary in main memory (a document map file is kept in main memory that contains document lengths needed for BM_25). There is clearly a trade off between the number of keywords and document lengths you keep in memory persistently against sets being retrieved and manipulated for Query service. Having to do I/O for list elements when weighting inverted lists could reduce the search performance dramatically. Four other groups in the TREC-8 Web Track registered faster time than ours, but three of them did not register the same level of retrieval effectiveness as us (one group retrieved no relevant documents).

DATABASE	**QP TIME SECS (LI)**	**SCALE-INV**		**SCALE-TXT**		**RATIO**	
		Bs1	**Bs10**	**Bs1**	**Bs10**	**Bs1**	**Bs10**
BASE1	0.027 (1.08)	-	-	-	-	-	-
BASE10	0.121 (1.03)	1.93	-	2.28	-	4.32	-
WT100g	1.616 (1.02)	1.27	0.655	1.65	0.721	57.9	13.4

Table 7-12. Query processing results for large web track search experiments

Measuring Scalability over both the text and Inverted file we found that the figures for BASE1 to WT100g were very good, while the corresponding figures from BASE10 to WT100g were acceptable (see table 7-12). The same pattern with respect to elapsed time ratios found with VLC2 *title only* runs was found with Web Track runs: BASE1 to the other collections yielded super -linear ratios, while BASE10 to WT100g yielded sub-linear ratios. From table 7-12 it can be seen that *title only* VLC2 queries yield better scalability from BASE1 to other collections than the Web Track queries: this is another effect of the memory requirements for Web Track queries stated above.

It should be noted that the Scalability/Ratio measurements from BASE1 to the other collections should be treated with some caution, since the memory requirements and communication overheads for search times are vastly different. It is not clear that parallelism brings much benefit on collections of BASE1's size. While these results are good there is clearly some scope for improvement, in particular the application of various query optimization techniques available (Hawking, 1998) at the cost of retrieval effectiveness.

7.8 SUMMARY AND CONCLUSION

The *DocId* partitioning method shows advantages over *TermId* partitioning on all measures and is clearly a good scheme for applying parallelism to information retrieval if such is needed. The results in this chapter confirm that our synthetic model can successfully determine which of the partitioning methods is better (see chapter 5, section 5.3.2). The synthetic model can also predict the relative performance difference between the partitioning methods to a great extent. However our empirical results show that *TermId* with parallel sort performance is nearer to *TermId* with sequential sort, whereas our synthetic model predicted that search on *TermId* with parallel sort would be nearer to *DocId*. The problem with communication in search would appear to be more significant in real life than in our synthetic model. Any generic conclusions we make with respect to theoretical or practical performance results are applicable only to term weighting models.

We proposed a hypothesis on search performance on parallel IR systems that suggests that query size is the main factor. The second part of our hypothesis is confirmed by the results on *DocId* partitioning. However technological factors in parallel search, namely the importance of sort in term weighting search, appear to be significant than information theoretic factors such as query size. The prediction in the synthetic model that a sequential sort would be a bottleneck is confirmed by empirical results.

The *TermId* scheme does not work in our framework and given the simplicity and widespread use of the cluster model we use, we do not see the scheme as being a viable one for probabilistic search for sequential query service. However the load balancing evidence discussed in section 7.6 supports any argument that concurrent query service using *TermId* partitioning may be useful. While the first part of our hypothesis on search performance put forward in section 7.2 may be correct for a single query, our evidence on load imbalance suggests it could be false in concurrent query service. The direction for any further research in this area is to examine how the inverted file partitioning methods affect the performance of parallel IR systems that offer concurrent query service in order to examine our hypothesis further.

Chapter 8

Index Update Theory And Results

8.1 INTRODUCTION

One of the most neglected areas in IR is the issue of servicing updates to inverted files. In most applications this is understandable given that some databases are only updated very infrequently: for example Dialog and DataStar have databases which are updated weekly, monthly quarterly or even yearly (Knight-Ridder, 1997). Searching has taken priority over update as a consequence. Insertions are usually done off-line and en-masse when no one is using the system, e.g. overnight. Such methods may not be suitable for systems where information is received at more frequent intervals and 24 hour access to this information is required, e.g. a news service, and there is no time when the system could be taken down and the updates serviced in a batch (Schiettecatte, 1996). Updating inverted files is very expensive and periodically requires the re-indexing of the whole database. It is therefore becoming increasingly important to examine the impact of update and query service on inverted files. To date IR systems with inverted files have not had to deal with Concurrency Control (CC). To the best of our knowledge the research presented here is the only published research on using CC mechanisms on inverted files (MacFarlane et al, 1996). The chapter is split into two parts: one discussing theoretical problems in the absence of CC and one describing practical experiments examining partitioning methods in the update task. The first part defines some of the problems by examining query processing in the presence of an incorrect or non-existent CC mechanism while updating the inverted file. Issues such as delays and availability are also examined. We attempt a paper solution to these problems by looking at popular CC mechanisms. We declare assumptions made and the scope of operations. In the second part we describe the experimental methodology for our practical experiments together with the data used in them. We then discuss our results for update, transaction and index reorganisation. A conclusion is given at the end.

8.2 ASSUMPTIONS

Throughout this chapter it is assumed that no deletions or updates are done on inverted files, only reads and insertions. This is because text databases tend to be archival in nature, hence exhibit dynamic behaviour that is that of growth rather than slight fluctuation in size or decrease in size. We also assume that an inverted file may be partitioned and stored on several different disks. The inverted file structure assumed is described in chapter 3, section 3.2.4.

8.3 SCOPE OF OPERATIONS

An insertion of a term results in one of two things; i) a new term, its posting list and position information is added to the inverted file; ii) for an existing term a posting is appended to that term's postings list, the position information is stored and the dictionary file entry is updated (the number of postings entry is incremented). A read done on a particular term results in; i) a message to indicate that there are no occurrences; ii) a posting list (with position information if that is required) is returned for *n* document id's in which that term occurs. The operation WRITE_TERM takes on the semantics of an insertion and the operation READ_TERM takes on the semantics of a read. The action on both READ_TERM and WRITE_TERM is to consult the dictionary file first, and then the postings file.

8.4 STATEMENT OF THE PROBLEM

The general problem of concurrency control arises because of possible conflicts between transactions operating at the same time on a database. In the context of IR this can best be illustrated by an example involving a more or less simultaneous query and addition of a new document to the database. Consider the scenario in fig 8-1 where an insertion transaction is interleaved with a query transaction that access the same data set. Our aim is to make these transactions have the same effect as if they where executed sequentially, i.e. they are serially equivalent (Coulouris et al, 1994). The insertion and query access the same terms.

TIME	INSERTION	QUERY
i	WRITE_TERM(term1, docid)	
i+1		postings$_1$ = READ_TERM(term1)
i+2		postings$_2$ = READ_TERM(term2)
i+3	WRITE_TERM(term2, docid)	
		answer = postings$_1$ ⊗ postings$_2$

Fig 8-1. Example concurrency control scenario

The problem in this scenario is that the query has done a read before the insertion has finished a write. In distributed systems this problem is known as a dirty read, which is caused because the READ_TERM operation on term2 in the query conflicts with the WRITE_TERM operation on term2 in the insertion. An easy answer would be to prevent the query from reading until the insertion is complete. However different operations ⊗ will produce different side effects, therefore a simple block may delay a query unnecessarily. The following section describes the effects of various operations.

8.5 OPERATIONS AND THEIR EFFECTS

Examples of an incorrect or non-existent CC mechanisms on the operations AND, OR, AND NOT, PLUS/DOT, ADJ, SAMES, LIMIT and MIXED are discussed below. Explanations of some of these operations are given in appropriate sections. It should be noted that the operations XOR, NAND, NOR and unary NOT are left out since they are not used in operational systems. These operations and their semantics are taken from Robertson and Walker (1995). A comparison of the effects is given at the end of the section.

8.5.1 ⊗ = *AND*

The side effect found where ⊗ = AND is that the query will fail to retrieve a relevant document because the postings list for term2 does not contain the identifier for the inserted document. To prevent this from happening the query must be excluded from reading information for term2 until the insertion has completed its action. The side effect is that of a false dismissal.

8.5.2 ⊗ = *OR*

The side effect will depend on whether position data is used by later operations. If no position data is involved, there is no apparent side effect where ⊗ = OR since the document identifier for the inserted term will be in the postings for term1. Therefore there is little point in blocking the read from term2, since it has no effect on the retrieval using that operation. To do so would delay the operation unnecessarily and result in a reduction in concurrency. We will name the side effect where an unnecessary block is made a false delay. However position information for term2 will not be included in any result set. Therefore any later operations on the sets that involve proximity operations may cause false dismissals, e.g. given the query {*information* ADJ {*science* OR *retrieval*}} where term1 = *science* and term 2 = *retrieval.*

8.5.3 ⊗ = *AND NOT*

This operation is asymmetric, therefore the side effect is dependent on the order of the reads. For example given the sets term1 = {1,2,3,4} and term2 = {1,2} before insertion (see fig 8-2);

INSERTION	QUERY	RESULT SET
	READ_TERM(term1)	{1,2,3,4}
WRITE_TERM(term1, 5)		{1,2,3,4,5}
WRITE_TERM(term2, 5)		{1,2,5}
	READ_TERM(term2)	{1,2,5}
	answer = {1,2,3,4} AND NOT {1,2,5}	{3,4}

Fig 8-2. Example interleaving with AND NOT

A document with an id = 5 is inserted in the inverted file. The correct documents are retrieved. The side effect could be a false delay since the read operations do not conflict with the writes, *because of the order in which they were done*. However if the order is changed (see fig 8-3);

INSERTION	QUERY	RESULT SET
	READ_TERM(term2)	{1,2}
WRITE_TERM(term1, 5)		{1,2,3,4,5}
WRITE_TERM(term2, 5)		{1,2,5}
	READ_TERM(term1)	{1,2,3,4,5}
	answer = {1,2,3,4,5}AND NOT {1,2}	{3,4,5}

Fig 8-3. Example interleaving with AND NOT; order reversed

the read and writes conflict because the id for document 5 is inserted in term2's set and not term1's set. The side effect is a false drop.

8.5.4 ⊗ = *PLUS*, ⊗ = *DOT*

These operations are generic functions for term weighting as per the probabilistic or vector space models (if you include normalisation in the latter). The result of the PLUS operator is a simple sum-of-weights. The result of the DOT operator is a dot-product, i.e. the sum of products of posting and set weights. The side effect where ⊗ = PLUS or DOT is that the weight for term2 will not be included in the postings set which could have the results; i) loss of weight for term2 drops the total weight down to an incorrect ranking or ii) the total weight for term2 drops below a stated threshold (as in the vector space model) and the document is incorrectly rejected. Therefore there are two side effects for the PLUS or DOT operators; either a rank drop or a false dismissal.

8.5.5 ⊗ = *ADJ*

The ADJ operator is used to find two terms that are adjacent to each other. The discussion of this operator is based on an ordered ADJ. We assume with the operator ADJ that term1 and term2 have matching position information on the insertion. Position information must be in both sets for a given term for any meaningful comparison to take place. Therefore since no match can be made where the position information for either term1 or term2 is absent, the resulting side effect is a false dismissal. It should be noted that while the operation is asymmetric the side effect is symmetric, i.e. no matter what term set is read in first, the result is still a false dismissal. Consider the queries {information ADJ retrieval} and {retrieval ADJ information}.

8.5.6 ⊗ = *SAMES*

The SAMES operator is used to find terms that occur in the same sentence. The assumption on ADJ position information applies to the SAMES operator. Unlike ADJ, the operator SAMES is symmetric. As with ADJ no match can be made where the position information for either term1 or term2 is absent, therefore the resulting side effect is a false dismissal. The same problems apply to operators which look for same paragraph or field.

8.5.7 ⊗ = *LIMIT*

A LIMIT operator is used to restrict the size of the output set size: users are not interested in viewing millions of documents. This operator is not explicitly used in search, but is used automatically when an AND operation is applied to two sets. In Robertson and Walker (1995) it is suggested that term1 LIMIT term2 is restricted to a search on term1, limited to items that satisfy term2. Therefore in strictly Boolean terms the LIMIT operator is identical to AND; therefore the side effect is identical, i.e. a false dismissal. While the operator is not symmetric, the side effect is.

8.5.8 *Mixed operations*

INSERTION	**QUERY**	**RESULT SET**
	READ_TERM(retrieval)	{1,2,3}
WRITE_TERM(retrieval,5)		{1,2,3,5}
WRITE_TERM(science,5)		{1,2,3,4,5}
	READ_TERM(science)	{1,2,3,4,5}
	READ_TERM(information)	{3,4}
WRITE_TERM(information,5)		{3,4,5}
	{1,2,3,4,5} AND NOT {3,4}	{1,2,5}
	{1,2,3} AND {1,2,5}	{1,2}

Fig 8-4. Example interleaving with MIXED operations

A query may contain more than two terms and may use any of the above operations, where such is legal, e.g. term1 ADJ (term2 OR term3) is valid, but term1 ADJ (term2 AND term3) is not. What is being considered here is the interaction of binary operations or nesting of binary operations. The complexity of the side effects for these nested binary operations could be considerable. We give an example interleaving in fig 8-4 with a query; *retrieval* AND (*science* AND NOT *information*). The sets for each of the terms before the interleaving are; *retrieval* = {1,2,3}; *science* = {1,2,3,4}; *information* = {3,4}.

There are a number of observations to be made on this interleaving. The most important is that the read on the query and write on the insertion conflict on the term "information"; as a consequence, the next-to-last result is incorrect. However this does not affect the final result since the error is masked by the AND operation. This is a by product of this particular query; other more complicated examples may actually reintroduce the problem. We can infer that the interaction between binary operations determines which side effects occur, if any. The interleaving will also have an effect, but is not just a characteristic of mixed operations. We cannot decide what the side effects will be by simply looking at the constituent binary operations.

8.5.9 *Comparison of operator effects*

Table 8-1 shows a comparison of the effects on the operators discussed above.

OPERATOR	*SYMMETRIC*	*SIDE EFFECT*	*COMMENTS*
AND	YES	False Dismissal.	-
OR	YES	False Delay. False Dismissal.	For unnecessary blocks. For position data only.
AND NOT	NO	False Drop, False Delay.	Depends on order of term insertion.
PLUS, DOT	YES	Rank Drop or False Dismissal	Side effect depends on threshold limit set (if any).
ADJ	NO	False Dismissal.	Side Effect is symmetrical.
SAMES	YES	False Dismissal.	-
LIMIT	NO	False Dismissal.	Side Effect is symmetrical.
MIXED	POSSIBLY	Any or all.	Side effects could be very complex.

Table 8-1. Comparison of operator effects

8.6 CONCURRENCY CONTROL MECHANISMS

It should be noted that the above are only a small number of simple scenarios. But it does give a flavour of some of the problems that may occur if an incorrect or non-existent CC mechanism is used. From the above we can deduce two very important facts. The first is that we need to vary the exclusiveness of blocks on term postings according to the operation. The second leads on from the first and suggests that the query model used will have implications for side effects. The concept of isolation levels much used in Database systems (Date, 1983) is regarded as useful.

Isolation is defined as the degree of interference a transaction can tolerate (Date, 1983). We can define an isolation level for queries in which non-conflicting situations occur, e.g. reads can be allowed on OR's thus stopping potential false delays. A further point is that blocks on terms that are popular (i.e. there is a high rate of retrieval on them) are likely to cause bottlenecks. Therefore the blocking granularity (the size of object being blocked) used for the postings file will be crucial not only in determining the retrieval performance, query throughput and system utilisation, but the CC mechanism performance as well. If the granularity is too large then unnecessary blocks will be made causing delays. The following discuss the three main CC techniques used in Distributed Systems (Colouris et al, 1994) in the light of the above. The mechanisms are: locking, optimistic CC and timestamp ordering.

8.6.1 *Locking*

The Lock method is the most common form of CC. The method works by setting a lock on a data item that blocks out other conflicting operations. What we tend to find is that concurrent reads are allowed but only one write is allowed at any given moment and a read may not be allowed while a write is being serviced. This is known as the many readers, single writer problem. Having reads which lock out each other is far too exclusive and reduces concurrency. The main advantage of the lock method is that it is better in environments where operations are predominately updates. The main disadvantage is deadlocks can occur (see below), a problem made difficult if more than one fragment of an inverted file is accessed.

The operation of Locks is simple. In the case of the scenario in section 8.4, fig 8-1, term1 is blocked by the insertion preventing the read on term1 in the query. The query is blocked on term1 until the insertion releases the lock on both the accessed terms in one go; the query can then retrieve both terms as normal with no conflict. However for queries with OR operations this could cause false delays. We remedy this problem by setting a level of isolation that allows queries with the operation OR to proceed without attempting to set a lock. A lock is released when the transaction has finished with that particular term. If a lock is set for a data item that already has a lock held on it, it delays that request until the lock is released. How do we resolve deadlocks in inverted files? Consider the following scenario in fig 8-5;

TIME	INSERTION$_1$	INSERTION$_2$
i	WRITE_TERM(term1, docid$_1$)	
i+1		WRITE_TERM(term2, docid$_2$)
i+2		WRITE_TERM(term1, docid$_2$)
I+3	WRITE_TERM(term2, docid$_1$)	

Fig 8-5. Example deadlock

Insertion$_1$ requests a lock on term1 and Insertion$_2$ requests a lock on term2. The problem occurs when Insertion$_2$ requests a lock on term1 and Insertion$_1$ requests a lock on term2. Both are now deadlocked and the data items are inaccessible until one or the other of the insertions is aborted (and re-started at a later time). There are a number of ways to decide which insertion to abort; i) abort the younger insertion to allow the older one to commit straight away; ii) choose an insertion that uses up the least machine cycles and abort that one. In the case of distributed deadlock leaf nodes need to reach some form of distributed agreement on which insertion should be aborted. Further consideration is needed for the Lock method including the use of hierarchic Locks and Lock promotion, particularly for mixed query operations.

8.6.2 *Optimistic CC*

The optimistic CC method takes a different view of blocking; it avoids it all together. Isolation levels are therefore not needed. An optimistic strategy is used which allows transactions to proceed irrespective of the effect unless a conflict is found. It should be noted that the other two methods, locks and timestamps use a pessimistic strategy. The method works by keeping a tentative version of a data item, while the transaction is being processed. The use of tentative versions allows the transaction to abort without the need to rollback the effect of a given operation. There are three phases to optimistic CC; i) read phase; data items are read in from disk and are put in tentative versions. This phase is never interrupted. The various operations such as writing is done on the tentative versions. ii) validation phase; After the read phase is complete the transaction is compared with other transactions and if any conflicts are found, a transaction is aborted. Otherwise the transaction proceeds to the write phase. iii) write phase; read only transactions can commit immediately while transactions with writes in them make their tentative versions permanent. There are two types of validation; forward and backward. Forward validation checks the current transaction with later transactions and works by comparing the write set of that transaction with the read set of later overlapping transactions. Backward validation checks the current transaction with earlier transactions and works by comparing the read set of that transaction to the write set of earlier overlapping transactions. The main advantage of the method is that it is fast in the presence of few conflicts. The disadvantage is that a substantial body of work may need to be repeated if there are many conflicts and starvation (non-service of a transaction) may occur with some transactions. This could have an effect on system utilisation and throughput. No deadlocks occur with the method.

Non conflicting operations (OR's) do not need to be validated against any other transaction. With backward validation a query or insertion is checked against earlier insertions; if any overlaps are found the query or insertion is aborted. With forward validation we compare an insertion with later queries or insertions and either abort the transaction being validated or any later transaction submitted to the system. We can see that in forward validation we have the option to either abort the transaction or a later one. Using backward validation we only have one choice, to abort the current transaction since earlier transactions have already committed. However forward validation is more complex than backward since it has to account for new transactions starting whilst still in the validation process. The practicality of using the method on the inverted file CC mechanism will depend on the update rate, i.e. the higher the update rate, the more chance of conflict and the less the method is useful. For situations where insertions are rare, optimistic CC could be useful.

8.6.3 *Timestamp ordering*

With this method a transaction is assigned a timestamp when it is initiated. The timestamp can take on a physical or logical value. The method works by comparing timestamps and if there are conflicts then a transaction is aborted. Each operation is validated as it is executed. There are three simple rules for transaction conflicts; i) To be able to write, a transaction must have the maximum read timestamp to prevent conflict on reads of other transactions; ii) To be able to write, a transaction must have the maximum write timestamp to prevent conflicts on writes of other transactions; iii) a transaction can only read a data item where the timestamp has a later value than the committed version to prevent conflicts on reads. We can allow for an isolation level in the event of a non-conflicting operation by ignoring timestamp comparison. The main advantage of the method it is that is better for environments where reads outnumber writes. The main disadvantage is that the timestamps determine the order of serialisation sequentially, according to the value assigned.

In IR systems the method is simple and would work as follows. A timestamp is compared only if the isolation level requires it. Operations where conflicts could occur have their timestamps checked as per the rules above. A problem could occur if an insertion has an older timestamp than a query. Because of rule i) we have a conflict and therefore have to abort the insertion. As with optimistic CC this could be problematic in certain applications.

8.6.4 *Comparison of CC methods*

From the above we have a number of choices for CC. The target application will determine which of the three would be suitable. If there are more queries than insertions, then we suggest that the Timestamp ordering method be used. If insertions are more frequent than queries, then we suggest that the Locking method be used. If there are few insertions, then the optimistic CC method could be useful. It is possible to use a more analytical approach to estimate the actual delays and overheads of each mechanism that would provide a more accurate comparison, but this is not attempted here. Such would include the cost of lock overheads, transaction rollback, space costs etc.

8.7 DELAYS AND AVAILABILITY

The requirements for document availability will depend on the application, e.g. a News Service or an On-Line Public Access (OPAC) system. The availability semantics would determine what CC mechanism to use. We could perhaps limit availability of documents to the log-on time, where any documents are unavailable if they are inserted during the session. In such a case no CC mechanism would be needed, since documents could be inserted overnight. This may not be suitable for a News Information Service application, where news from around the world may be required when it is received. We term these semantics as *Log-On availability* semantics. If we take the semantics that do not make documents available until the last write in the insertion is complete, then many of the mechanisms described above would not be needed since a false dismissal would not be deemed to have occurred. We will call such semantics *Late Availability.* However other side effects cannot be so easily ignored. To allow queries and insertions to go ahead without blocking could lead to false drops or rank drops. Therefore some blocking will be needed in *Late Availability* semantics. This has the unfortunate side effect of delaying some queries for a period, when such blocks are needed. If we take semantics that make documents available when the first write in the insertion is completed, a full set of the mechanisms described above would be required. We will call such semantics *Early Availability.* Any query which shares terms with insertions will need to be delayed, an undesirable effect in some applications. By its nature the inverted file technology gives priority to queries. This is because inverted file search is comparatively cheap while inverted file update is very expensive. Searching speed is the reason that inverted files have become the dominant technology in IR. Therefore the assumptions made in this chapter may not be sustainable for many applications. However certain side effects such as rank drops or false drops would be very undesirable in an IR system. To prevent these some delays may be needed. It is a question of determining how far search is offset against insertion.

Another important factor that could have a dramatic impact on performance is unnecessary processing. This could occur with *Early Availability* semantics because of the possibility of starvation of queries. Transactions may be aborted and re-started a number of times if one term or group of terms is being updated constantly for a time period (e.g. a spurt of News on a particular subject).

From the above it can be seen that we have a number of conflicting requirements. We want to make documents available as soon as possible, without delaying queries unduly. We do not want the occurrence of multiple aborts. Such choices come about because of the "black and white" nature of *Late* and *Early Availability* semantics, which take rather an extreme viewpoint on document availability. We need to find another form of availability semantics that does not introduce too many unnecessary delays and does not do too much unnecessary processing. It should be noted that we are unlikely to be able to find a system of semantics in which no delays occur and unnecessary processing is never done.

8.8 METHODOLOGY FOR PRACTICAL EXPERIMENTS

We have stated in the introduction to this thesis that we examine the issue of one job at a time service, e.g. one transaction at a time. The main reason is the viability of designing and building a complete transaction processing system within the time available. Another was the practicality of examining retrieval effectiveness problems on concurrent transaction service as examined theoretically above. The basic problem is that we do not have an evaluation method. We could consider using such metrics as recall and precision, but these are unlikely to be useful except in extreme situations. Servicing one transaction at a time allows us to ignore this problem at the cost of restricting the level of parallelism on update transactions.

Much of the previous work in the area of inverted file maintenance (Reddaway, 1991; Shoens et al, 1994; Clarke and Cormack, 1995) have advocated the use of buffering updates to save on I/O. Some argue that to update the index for each individual arriving document is inefficient (Shoens et al, 1994) but use a synthetic workload performance analysis to support their arguments. We attempt to simulate a persistent service for updates without coding a complete transaction service, accepting that it is better to wait a little before updating an index. To do this we keep an in-core buffer to which updates are added when they are received. When this buffer is full, we initiate an index reorganisation merging the in-core buffer index with the index kept on disk. In order to do this we use the following strategy:

i) Read in inverted list from disk.
ii) Add new postings to inverted list.
iii) Save the new postings to disk to a temporary postings file.

As we are unlikely to be able to keep the dictionary in-core, we keep a subset of the keywords in memory, with each element of the subset a header of a keyword block held on disk. All hit keyword blocks are saved to a temporary Keyword file for realism. The advantage of this method is that we can do a realistic disk re-organisation simulation without the need for expensive rollbacks in order to conduct repeated experiments on the same data set. We do not attempt to reorganise the whole index as we assume that a large chunk of the database will never be referenced by incoming updates. The transaction we refer to as updates are collection updates or document insertions. Our priority is to try and keep the index in a state which will allow the search process to service queries as fast as possible. We do however allow the service of transactions while the reorganisation of the index is being done. There is a strict interleaving between the reorganisation of a term and transaction service to prevent concurrency problems examined above. There may therefore be some delays to transactions while a reorganisation of the index is being done. The document availability semantics we use is *Late Availability* which is defined in section 8.7 above.

8.9 DATA AND SETTINGS USED

Transaction set Name	No of Updates	No of Queries	No of Transactions
UPDATE1	40	400	440
UPDATE2	80	400	480
UPDATE3	200	400	600
UPDATE4	400	400	800
UPDATE	500	400	900

Table 8-2. Details of transaction sets used in experiments

The data used in the experiments was the BASE1 and BASE10 collections (see chapter 3, section 3.3.1). We use both types of builds for indexes: *distributed builds* and *local builds*. For the *distributed build* method we use the BASE1 collection only, creating indexes on 1 to 7 nodes and servicing transactions on all of those indexes. Two types of index where built for these experiments: one set using *TermId* partitioning and one using *DocId* partitioning. The BASE1 and BASE10 collections were used for the *local build* method, running queries on 8 nodes: the client and top node had to be placed on the same node as one leaf nodes. The *DocId* partitioning method is used on these experiments. We built one set of indexes which contained position data (POS) and one set without position data (NPOS) for both types of build methods and both types of partitioning methods.

Table 8-2 shows the transaction sets used in our experiments and the balance between query and update transactions. The queries are based on topics 1 to 450 of the TREC1 to TREC8 ad-hoc tracks: 400 queries in all (the topics 201-250 in TREC4 did not have a title field in the topics). The terms were extracted from TREC topic descriptions using an Okapi query generator utility to produce the final query. The average number of terms per query is 3.46. The document updates were chosen from a Reuters-22178 collection (Lewis, 1997b) not in the VLC2 set: we refer to this file as REUTERS. We chose this set because can guarantee that the data is new to the VLC2 set. The REUTERS file is 1.2 Mb in size and has 1000 documents (or records). We took both these sets and created transaction sets with differing numbers of updates and queries, varying the number of updates to queries, ranging from 10 queries per 1 update down to more than one update per query.

We apply these transactions to all the indexes built (described above) both in the presence and absence of an index reorganisation. Where no index reorganisation is done during transaction processing we initiate one at the end of transaction service and record the time. This allows us to both examine the relationship between updates and queries in transactions as well as finding a good point where buffer re-organisation is needed. In order to examine the effect of index reorganisation we fill up buffers with 500 documents from the REUTERS document set and initiate a re-organisation before starting transaction processing. All transaction processing figures produced are averages of 5 runs per experiment. For the one leaf nodes experiments we use a client/server process. We used the CF term allocation policy for nodes when using *TermId* partitioning (see chapter 4, section 4.4.2 for details). We use a number of measures to examine the results (these metrics and our requirements of them are declared in chapter 3, section 3.4.1). These are elapsed time in seconds, load imbalance (LI), speedup/efficiency, and scalability for transactions and index re-organisation and transaction throughput (transactions per hour).

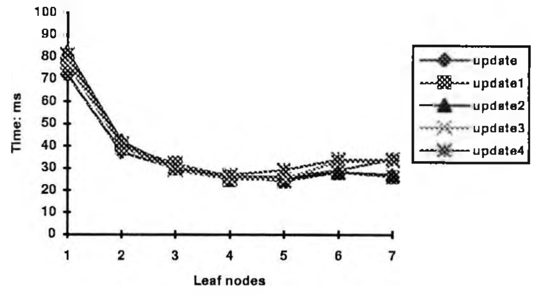


Fig 8-6. BASE1 [*DocId*]: average elapsed time in ms for update transactions (postings only)

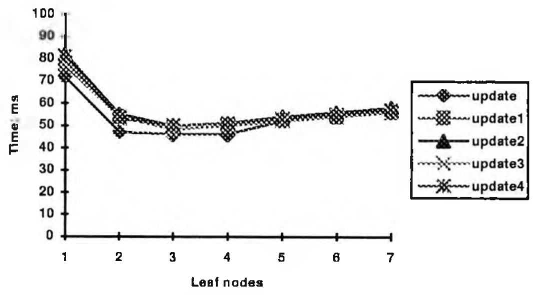


Fig 8-7. BASE1 [*TermId*]: average elapsed time in ms for update transactions (postings only)

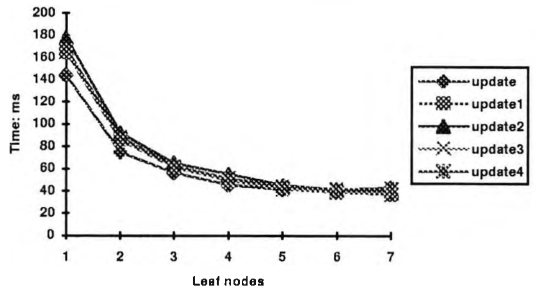


Fig 8-8. BASE1 [*DocId*]: average elapsed time in ms for update transactions (position data)

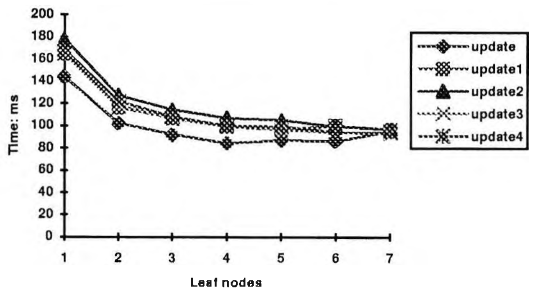


Fig 8-9. BASE1 [*TermId*]: average elapsed time in ms for update transactions (position data)

8.10 EXPERIMENTAL RESULTS ON TRANSACTION PROCESSING

We have a number of aspects which we wish to examine by looking at the empirical results produced. The first of these is the issue of update performance (see section 8.10.1). Is there a big performance penalty in only allowing one update at a time in the system thereby restricting parallelism for that transaction type? We also need to examine the transactions as a whole looking at aspects such as the interaction between queries and updates and its impact on performance (see section 8.10.2). Both updates and transactions are examined in the presence and absence of index reorganisation. The performance of index reorganisation is examined in section 8.10.3, together with a discussion on a good buffer size for the collections being examined. A summary of the experimental results is given in section 8.10.4.

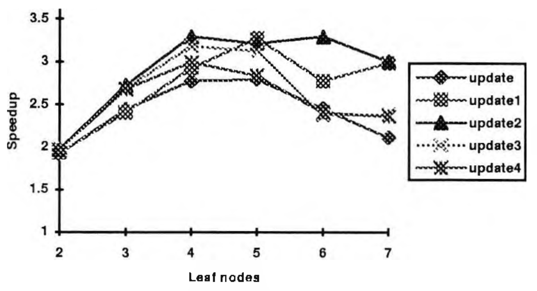


Fig 8-10. BASE1 [*DocId*]: speedup for update transactions (postings only)

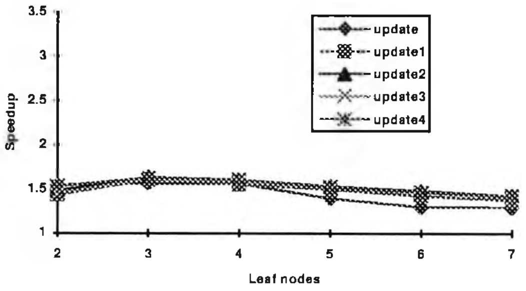


Fig 8-11. BASE1 [*TermId*]: speedup for update transactions (postings only)

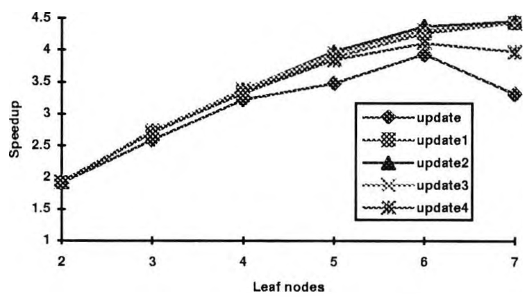


Fig 8-12. BASE1 [*DocId*]: speedup for update transactions (position data)

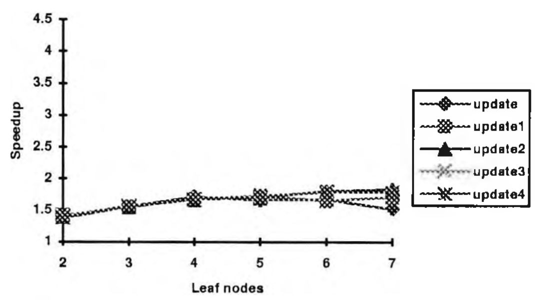


Fig 8-13. BASE1 [*TermId*]: speedup for update transactions (position data)

8.10.1 *Performance of Update Transactions*

As there is no general criterion for response time for update transactions as there is for query transactions (Frakes, 1992) we need to define one here. The criterion we use is that updates should be done within 1/10th of a second (or 100 milliseconds). This strict criterion is chosen because we want to ensure that queries are not delayed much, although users who submit documents for update would prefer a fast response. The elapsed time for update transactions is quite small for most runs (see figs 8-6 to 8-9). All times are under 100 milliseconds and times do reduce with increasing numbers of leaf nodes. There are two main observations from this. The first is that update transaction elapsed times meet our criterion and are therefore acceptable in our terms. Any delays by blocking other transaction while an update is done are therefore small. The second is that speedup is found in systems using parallelism, which is surprising given the restrictions on parallelism with the type of update transaction processing implemented (see figs 8-10 to 8-13: efficiency figures are given in appendix A4). The results show that *DocId* partitioning has a much more beneficial effect on elapsed times than *TermId* partitioning and the advantage in elapsed time using multiple leaf nodes is superior with *DocId*. The reasons for these effects are twofold: memory and communication. With *DocId* the increase in memory affects elapsed time positively, and communication is done with one leaf node only. This memory advantage is offset with extra communication with *TermId* as document data must be communicated to all leaf nodes. It should be noted that most of the conclusions drawn here apply to updates which record position data. The exception is that *TermId* partitioning in many cases does not meet the 100 millisecond criterion together with the single leaf nodes run (see fig 8-9).

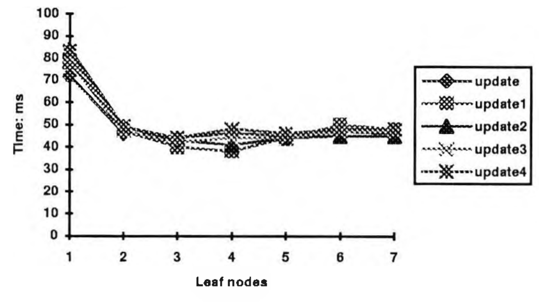


Fig 8-14. BASE1 [*DocId*]: average elapsed time in ms for update transactions during index reorganisation (postings only)

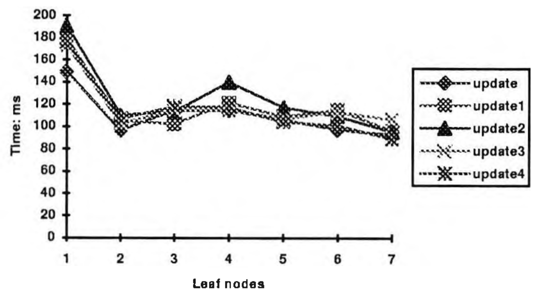


Fig 8-16. BASE1 [*DocId*]: average elapsed time in ms for update transactions during index reorganisation (position data)

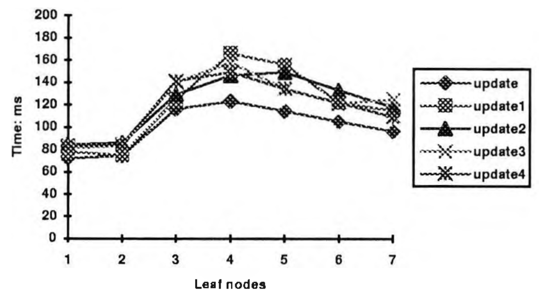


Fig 8-15. BASE1 [*TermId*]: average elapsed time in ms for update transactions during index reorganisation (postings only)

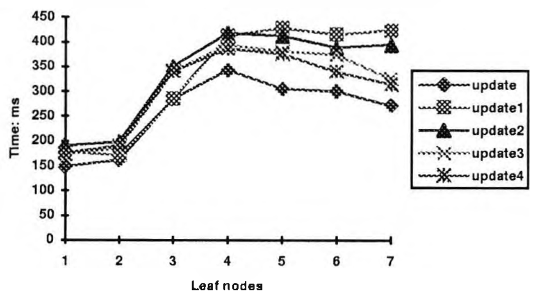


Fig 8-17. BASE1 [*TermId*]: average elapsed time in ms for update transactions during index reorganisation (position data)

Figs 8-14 to 8-17 shows the effect of initiating an index reorganisation while serving update transactions. Elapsed times on both types of partitioning method are increased, but *DocId* partitioning is much better able to handle the resource contention than *TermId.* In terms of our 100 millisecond criterion, *DocId* meets our requirement while *TermId* partitioning does not. While *DocId* runs show reduction in elapsed time over multiple leaf nodes, *TermId* runs actually record a reduction in performance. The reason for this is simple: index reorganisation on *DocId* partitioned inverted file is done on much shorter lists. Therefore a request for transaction service on *TermId* partitioning is more likely to be delayed, hence the increase in percentage terms for elapsed time over *DocId* as shown in figs 8-18 to 8-21. With respect to indexes which contain position data, most runs apart from a few on *DocId* partitioning exceed the 100 millisecond criterion. *TermId* partitioning runs are particularly badly affected with some runs registering an increase of around 300% over elapsed times when index reorganisation is done.

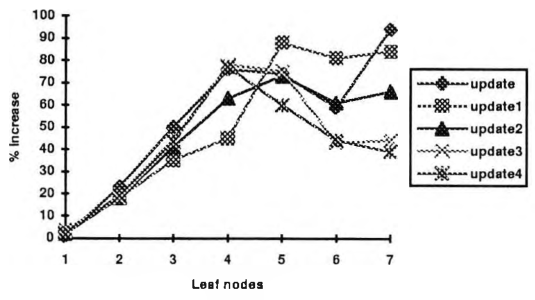


Fig 8-18. BASE1 [*DocId*]: % increase in average elapsed time for update transactions during index reorganisation (postings only)

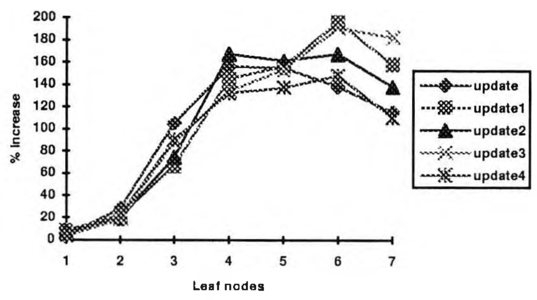


Fig 8-20. BASE1 [*DocId*]: % increase in average elapsed time for update transactions during index reorganisation (position data)

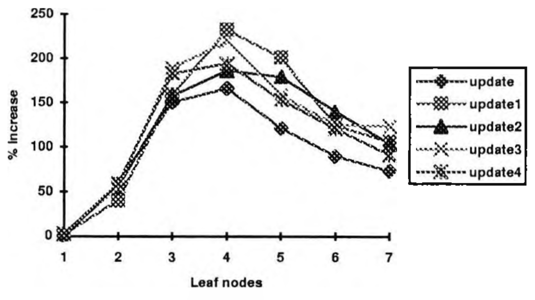


Fig 8-19. BASE1 [*TermId*]: % increase in average elapsed time for update transactions during index reorganisation (postings only)

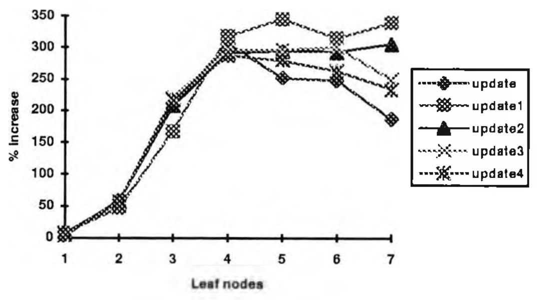


Fig 8-21. BASE1 [*TermId*]: % increase in average elapsed time for update transactions during index reorganisation (position data)

Metric	Collection	UP	UP1	UP2	UP3	UP4
Postings Only (No Positions)						
Elapsed Time (ms)	BASE1 BASE10	43 109	43 124	46 124	40 121	43 123
Scalability	BASE10	3.97	3.46	3.72	3.33	3.51
Elapsed Time (ms) during index update	BASE1 BASE10	55 268	64 380	62 359	60 310	57 299
Scalability during index update	BASE10	2.07	1.67	1.73	1.95	1.91
Position Data						
Elapsed Time (ms)	BASE1 BASE10	52 202	48 265	51 261	48 243	51 246
Scalability	BASE10	2.55	1.83	1.97	1.98	2.06
Elapsed Time (ms) during index update	BASE1 BASE10	103 621	130 971	131 975	125 892	115 817
Scalability during index update	BASE10	1.66	1.34	1.34	1.40	1.40

Table 8-3. BASE1/BASE10 [*DocId*]: index update results for update transactions

Table 8-3 shows the details of comparable BASE1 and BASE10 runs using the *DocId* partitioning method. It should be noted that BASE10 runs are slightly higher than our criterion for elapsed times for update. It may not therefore be possible to set such a strict criterion for larger databases, and we may have to relax our requirements to say, a second. All BASE10 elapsed times are under a second, even updates done on indexes with position data while an index reorganisation is being done. The scalability for update transactions on the BASE10 collection is very good indeed, particularly for indexes with postings only data. The scalability reduces while index reorganisation is being done, but is still good.

8.10.2 *Performance of Transactions as a whole*

The average elapsed time for transactions as a whole is very good with all times under a second, including BASE10 experiments. Figs 8-22 to 8-25 shows average elapsed times for transactions on the BASE1 collection using all types of indexes and partitioning methods.

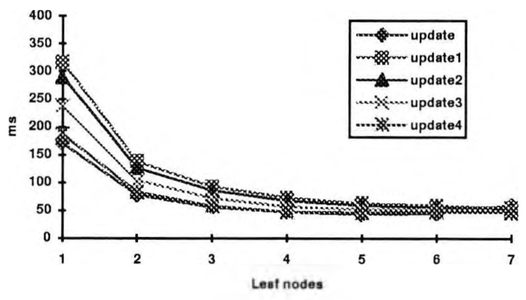


Fig 8-22. BASE1 [*DocId*]: transaction average elapsed times in ms (postings only)

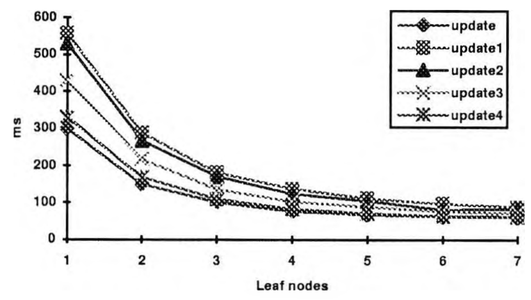


Fig 8-24. BASE1 [*DocId*]: transaction average elapsed times in ms (position data)

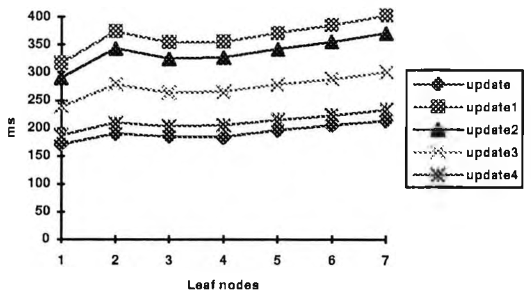


Fig 8-23. BASE1 [*TermId*]: transaction average elapsed times in ms (postings only)

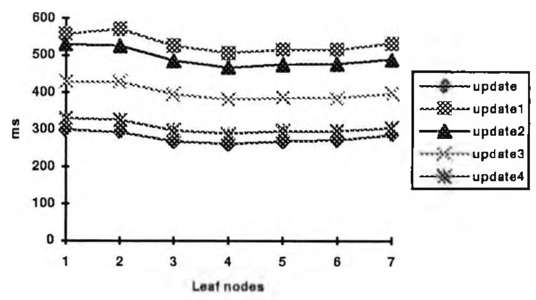


Fig 8-25. BASE1 [*TermId*]: transaction average elapsed times in ms (position data)

From these elapsed times it can be seen that there is a reduction in average time when the number of update transactions is increased and when *DocId* partitioning is used. The reduction due to increased level of updates is because updates are smaller in average time and will reduce the average transaction time. The *DocId* partitioning method outperforms *TermId* quite considerably on any of the transaction sets used. The performance problem found with runs on *TermId* partitioning in previous experiments (see chapter 7 on probabilistic search) severely effect the overall performance of those runs. No real speed advantage by the use of parallelism is demonstrated in any of the *TermId* partitioning experiments. In fact slowdown is registered for all parallel runs on indexes with postings only data (see fig 8-27). Speed advantage on indexes containing position data is recorded, but is very slight (see fig 8-29).

With *DocId* partitioning we do gain speed advantage using parallelism (see figs 8-26 and 8-28), but the proportion of updates in the transaction set may actually increase the average elapsed time when more leaf nodes are used (see figs 8-22 and 8-24). The level of parallelism which can be successfully deployed depends on the balance in time between updates and queries, at the point where gain in parallelism is outweighed by loss in servicing updates. Parallel efficiency figures are given in appendix A4.

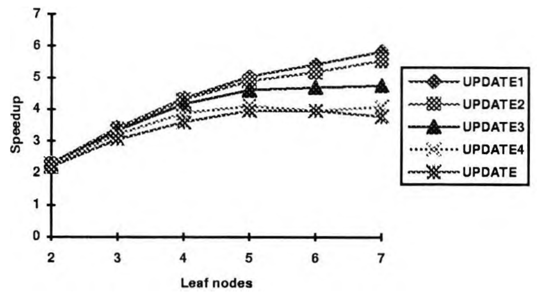


Fig 8-26. BASE1 [*DocId*]: speedup for all transactions (postings only)

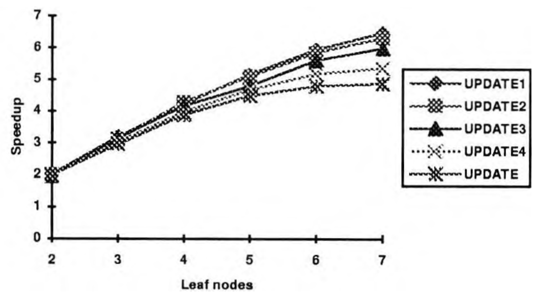


Fig 8-28. BASE1 [*DocId*]: speedup for all transactions (position data)

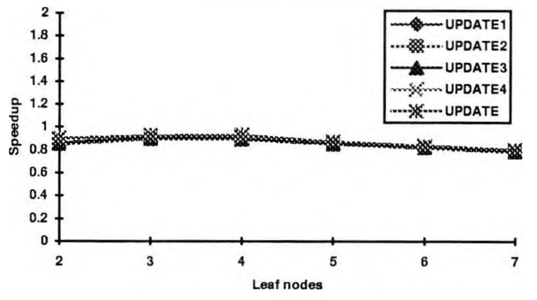


Fig 8-27. BASE1 [*TermId*]: speedup for all transactions (postings only)

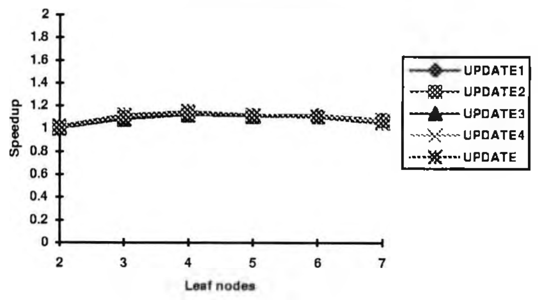


Fig 8-29. BASE1 [*TermId*]: speedup for all transactions (position data)

Figs 8-30 to 8-33 shows the effect of index reorganisation on transactions serviced over BASE1 collection. Percentage increases in elapsed time for the same data can be found in appendix A4. The results show that *DocId* partitioning outperforms *TermId* if an elapsed time criterion is used. While runs on *DocId* partitioning using parallelism reduce run times over the client/server runs, *TermId* runs actually increase in time. This evidence is consistent with the update transaction results described above. However it is clear that *DocId* partitioning after a certain parallel machine size holds the run times constant, and the ability to cope with resource contention is far superior to that of *TermId*. There is some doubt as to the wisdom of deploying parallelism after a given point, but other factors such as the total time for an index reorganisation are important. Our choice of either parallelism or the actual level of parallelism will depend on the balance between normal transaction processing and transaction processing during an index update. A further interesting observation is that transaction sets with more update transactions are less affected by resource contention than others with more query transactions, which is particularly noticeable in *TermId* results (see fig 8-31 and 8-33). The reason for this is that update transactions are faster than query transactions and are therefore much less affected when the index is being updated.

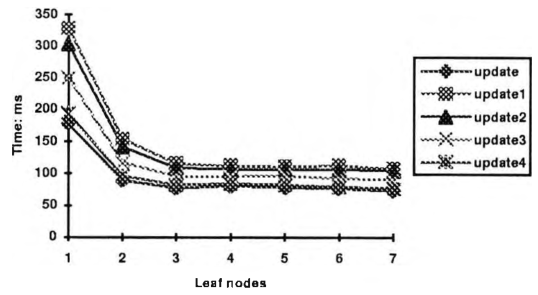


Fig 8-30. BASE1 [*DocId*]: average elapsed time in ms for all transactions during index reorganisation (postings only)

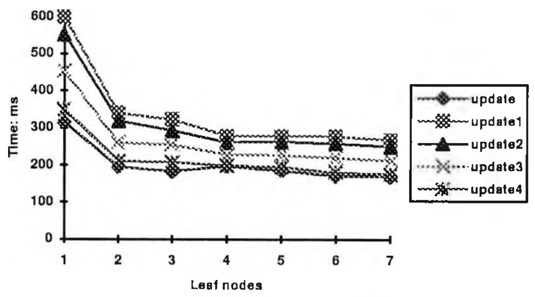


Fig 8-32. BASE1 [*DocId*]: average elapsed time in ms for all transactions during index reorganisation (position data)

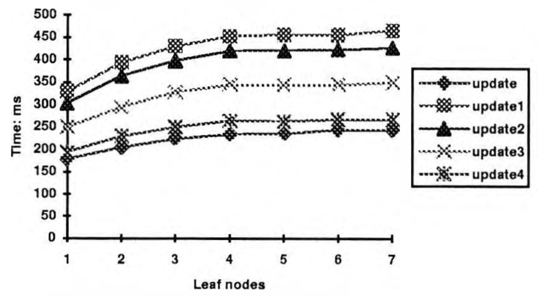


Fig 8-31. BASE1 [*TermId*]: average elapsed time in ms for all transactions during index reorganisation (postings only)

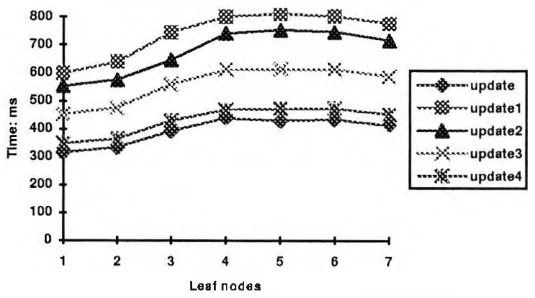


Fig 8-33. BASE1 [*TermId*]: average elapsed time in ms for all transactions during index reorganisation (position data)

What effect do these results have on throughput? In figs 8-34 to 8-38 the throughput figures are declared, with the data separated into transaction sets. The suffix "ro" in the diagrams signifies that the run was done in the presence of an index reorganisation. The throughput measure is thousands of transactions per hour

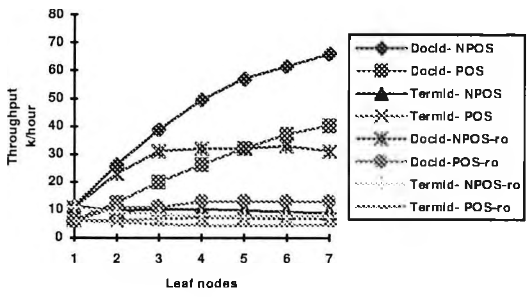


Fig 8-34. BASE1: combined transactions throughput for UPDATE1 transaction set

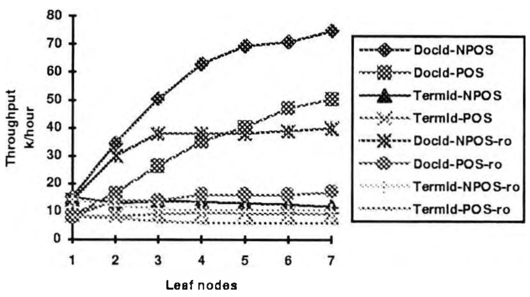


Fig 8-36. BASE1: combined transactions throughput for UPDATE3 transaction set

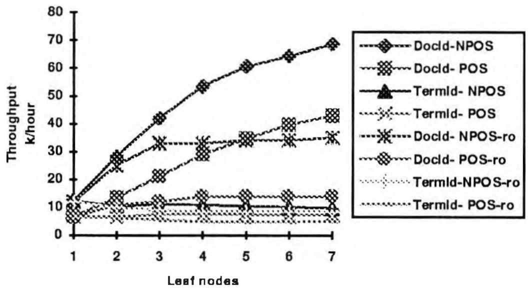


Fig 8-35. BASE1: combined transactions throughput for UPDATE2 transaction set

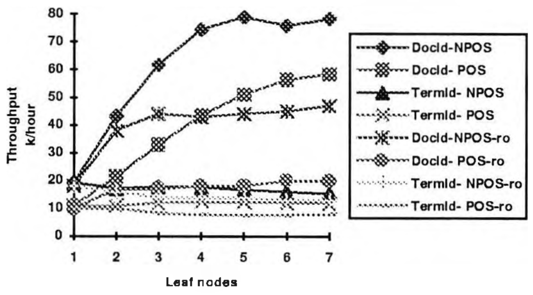


Fig 8-37. BASE1: combined transactions throughput for UPDATE4 transaction set

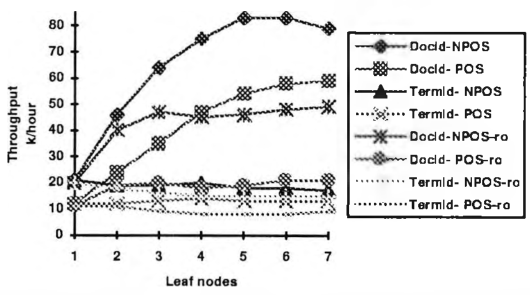


Fig 8-38. BASE1: combined transactions throughput for UPDATE transaction set

The main conclusion from these throughput results is that *DocId* partitioning outperforms *TermId* using any type of index (as would be expected from the elapsed time data). Using this measure demonstrates how disappointing the performance of *TermId* actually is: throughput is not improved by the addition of extra leaf nodes. Many runs are limited to a throughput of 20k transactions per hour. The best performing index type/partitioning pair is *DocId* with postings only indexes on any of the transaction sets. It can be seen in the diagrams through *DocId* with postings only data, that the transaction set has an impact on trends in throughput. For example on the UPDATE1 set there is a clear increase in throughput for increasing numbers of leaf nodes, while throughput on the UPDATE set shows a clear tailing off effect with larger numbers of leaf nodes (see figs 8-34 and 8-38). Throughput on the index type/partitioning method relative to each other is consistent irrespective of the transaction set under scrutiny.

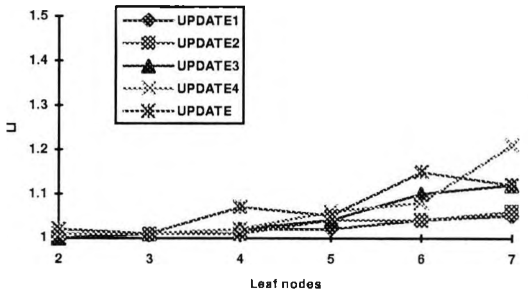


Fig 8-39. BASE1 [*DocId*]: load imbalance for all transactions (postings only)

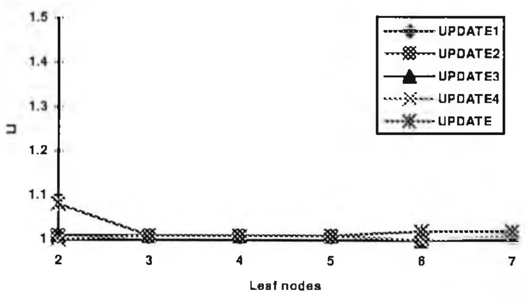


Fig 8-41. BASE1 [*TermId*]: load imbalance for all transactions (postings only)

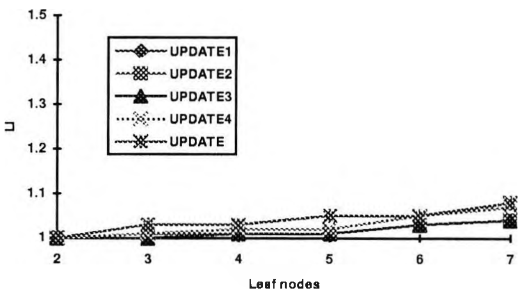


Fig 8-40. BASE1 [*DocId*]: load imbalance for all transactions (position data)

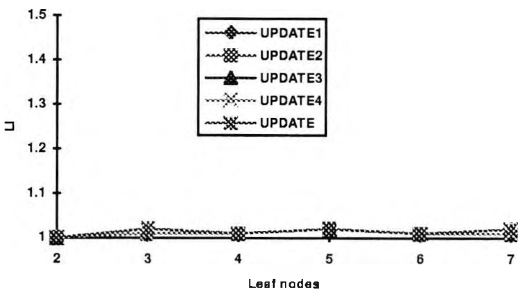


Fig 8-42. BASE1 [*TermId*]: load imbalance for all transactions (position data)

How does load imbalance affect the results given above? Load imbalance does not appear to be a significant problem: figs 8-39 to 8-42 show the overall level of load imbalance for all transactions. It can be seen that imbalance is higher in *DocId* than it is in *TermId*, for both types of indexes. The imbalance figures for all results are relatively small, but clearly there is an increase in imbalance with increasing parallel machine size on *DocId*, while imbalance on *TermId* remains fairly constant. The key result here is that document updates do not harm overall load imbalance significantly. The round robin method of distributing document updates to nodes when *DocId* partitioning is used is a reasonable method. The results also show that it may be possible to offer better concurrent transaction service on *TermId* partitioning than *DocId* partitioning (this is consistent with imbalance results found in probabilistic search - see chapter 7).

Table 8-4 shows the scalability results for all transactions. Average elapsed times for BASE10 runs during normal transaction processing are all under half a second when postings only indexes are used and under a second for position data indexes. The delays on BASE10 while indexes are updated are considerable and runs are over double, a factor particularly significant for indexes with position data. It may not be viable to use the index update method for this task, particularly if queries are delayed beyond the 10 second elapsed time recommendation (Frakes, 1992) during an index reorganisation on much larger collections. Scalability is very good and increases with the number of updates in a transaction: as would be expected since updates provide much better scalability than queries (see table 8-3 above). Elapsed times trends are inverse to that of scalability and for the same reason.

Metric	Collection	UP	UP1	UP2	UP3	UP4
Postings Only (No Positions)						
Elapsed Time (ms)	BASE1 BASE10	60 257	75 479	73 440	66 363	60 280
Scalability	BASE10	2.35	1.56	1.66	1.83	2.14
Elapsed Time (ms) during index update	BASE1 BASE10	80 551	118 1021	112 949	98 776	82 604
Scalability during index update	BASE10	1.46	1.15	1.18	1.26	1.36
Position Data						
Elapsed Time (ms)	BASE1 BASE10	72 448	103 891	97 818	84 660	73 505
Scalability	BASE10	1.61	1.15	1.18	1.27	1.45
Elapsed Time (ms) during index update	BASE1 BASE10	159 1368	265 2562	246 2364	205 1924	167 1517
Scalability during index update	BASE10	1.17	1.04	1.04	1.07	1.10

Table 8-4. BASE1/BASE10 [*DocId*]: index update results for all transactions

It is clear that within our experimental framework the best partitioning method for transaction processing is *DocId.* Both the experiments discussed here and work discussed throughout this thesis show that *DocId* partitioning provides better performance both in normal transaction processing and when an index reorganisation is initiated for all types of transactions. However the imbalance figures demonstrate that concurrent transaction processing might work well on *TermId* partitioning, a conclusion which reinforces our previous experience with search (see chapter 7). Scalability of transactions using *DocId* partitioning is good, but results demonstrate that the index update task defined here may not be a viable solution for much larger collections than ones considered here.

8.10.3 *Performance of Index Reorganisation*

The results found in our index reorganisation performance confirm that it is better to wait for a given period and do the reorganisation collectively than do it on a one document basis (Shoens et al, 1994). Figs 8-43 to 8-46 shows the index reorganisation results using elapsed time in seconds.

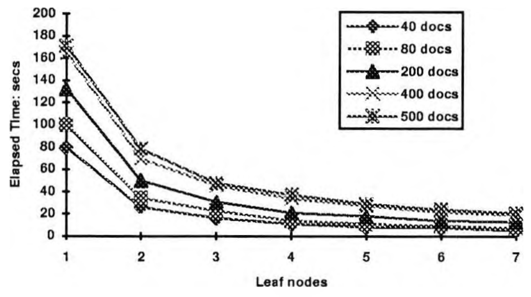


Fig 8-43. BASE1 [*DocId*]: index reorganisation elapsed time in seconds (postings only)

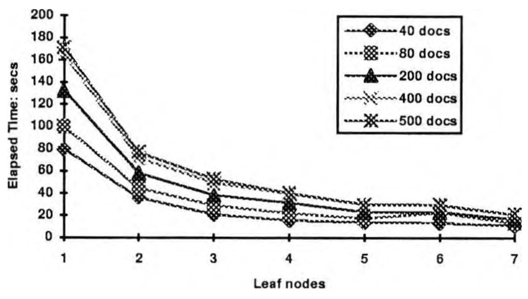


Fig 8-45. BASE1 [*TermId*]: index reorganisation elapsed time in seconds (postings only)

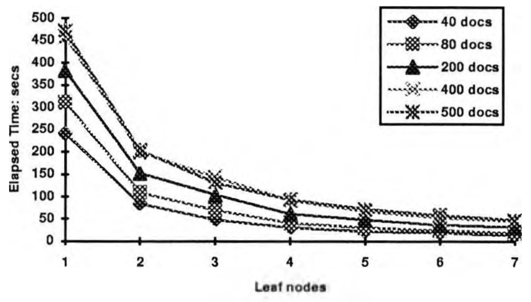


Fig 8-44. BASE1 [*DocId*]: index reorganisation elapsed time in seconds (position data)

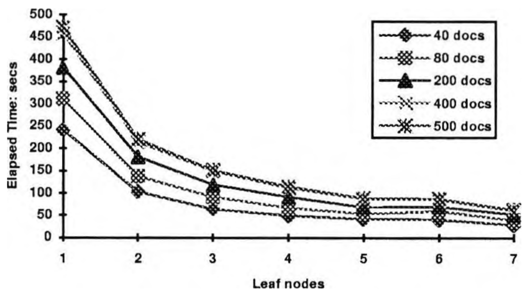


Fig 8-46. BASE1 [*TermId*]: index reorganisation elapsed time in seconds (position data)

Above all these figures show how expensive index reorganisations are particularly for indexes with position data. It should be noted that these figures are much reduced from a method which would require a reorganisation of the whole index. The best buffer size for this data is 500 documents: there is very little difference between reorganisations done on buffer sizes of 500 document and 400 documents, particularly for indexes with position data. There is an increase in the elapsed time for increasing buffer size on all runs, but the increase is not linear with the number of documents in the buffer: the results on multiple leaf nodes are the same. Comparing the partitioning methods, elapsed times on *DocId* are better than *TermId* using all buffer sizes and on all multiple leaf nodes runs apart from 2 leaf nodes on a 500 document buffer. Speed advantage is shown in both partitioning methods by increasing the leaf nodes set in a run. Figs 8-47 to 8-50 show the speedup for index reorganisation on both partitioning methods.

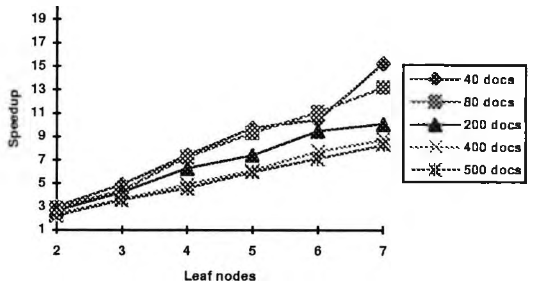


Fig 8-47. BASE1 [*DocId*]: index reorganisation speedup (postings only)

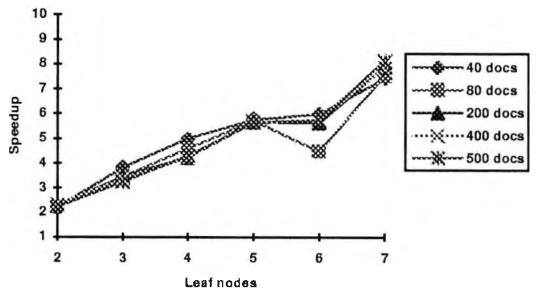


Fig 8-49. BASE1 [*TermId*]: index reorganisation speedup (postings only)

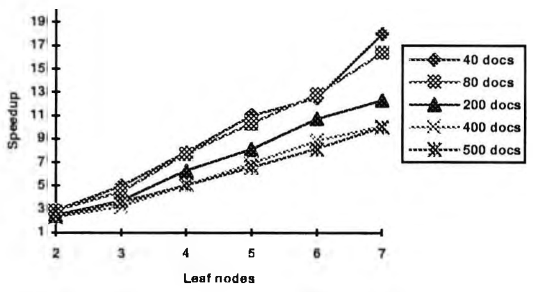


Fig 8-48. BASE1 [*DocId*]: index reorganisation speedup (position data)

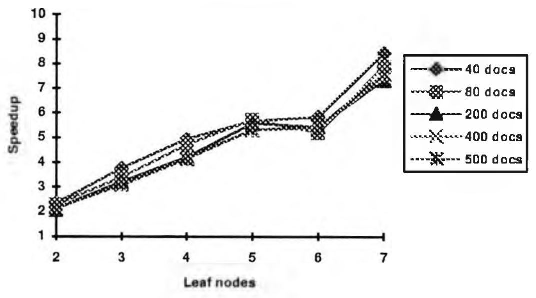


Fig 8-50. BASE1 [*TermId*]: index reorganisation speedup (position data)

Good speed advantage is shown by any number of leaf nodes using any type of partitioning method. Super-linear speedup is shown on both partitioning methods apart from *TermId* on any run using 6 leaf nodes with any type of index. The run on 6 leaf nodes on a 80 document buffer is particularly disappointing considering the other results. We will return to this factor when discussing load imbalance below. Efficiency figures are also very good and can be found in appendix A4. Why does this super-linear speedup and efficiency occur? If we examine the total time needed for an index reorganisation we find that all parallel runs reduce the total time for an index reorganisation, but with *DocId* partitioning there is a noticeable trend downwards with increasing numbers of leaf nodes (the data for this can be found in appendix A4). Figs 8-51 to 8-52 show the underlying reason why the super-linear speedup occurs.

The total number of posting records handled for *DocId* actually reduces with increasing numbers of leaf nodes, but *TermId* runs move much the same amount of data. The reason for this effect in *DocId* partitioning is that as the number of leaf nodes is increased, the more frequent terms which both the buffer and index shared are spread over more blocks which have fewer records associated with them. The more frequent occurring terms are interspersed among less frequent terms as more blocks are handled. This effect does not happen on *TermId* partitioning as much the same blocks will be handled by parallel runs of any leaf node size. There is some variation in *TermId* but the effect is minimal. Note that the number of postings moved for a 500 document buffer is always slightly more than those for a 400 document buffer. The total number of postings in BASE1 collection is 22.6 million: just over half the index is reorganised for just 500 documents reducing with increasing numbers of leaf nodes. The evidence suggests that a good buffer size for this data is 500 documents.

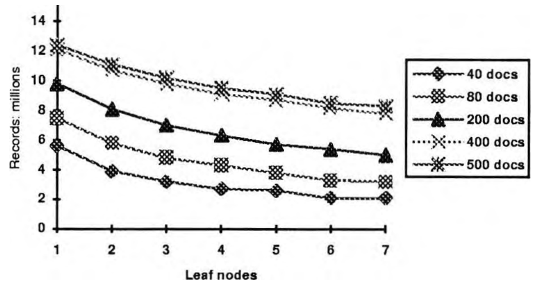


Fig 8-51. BASE1 [*DocId*]: millions of postings handled during index reorganisation

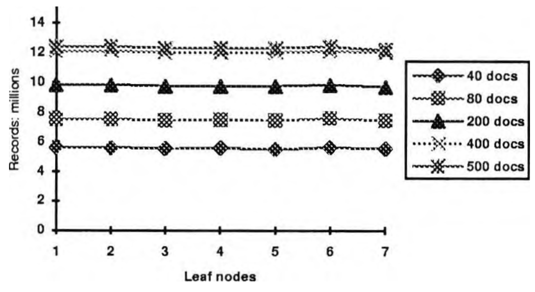


Fig 8-52. BASE1 [*TermId*]: millions of postings handled during index reorganisation

From the evidence given above there is clearly an offset between the advantage gained in *DocId* partitioning by increasing the number of leaf nodes and improvements in performance gained by waiting until the buffer has reached a given size. We can therefore make a case for delaying the initiation of index reorganisation on more leaf nodes until their buffers contain more documents. In this way we can take advantage of both effects discussed, i.e. less data movement on more leaf nodes and less time when an index update is being done. Figs 8-53 to 8-56 provides more evidence of the increased buffer effect on load imbalance.

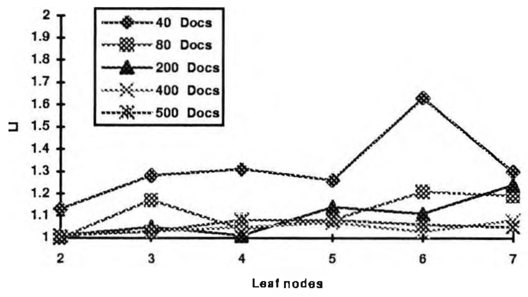


Fig 8-53. BASE1 [*DocId*]: index reorganisation load imbalance (postings only)

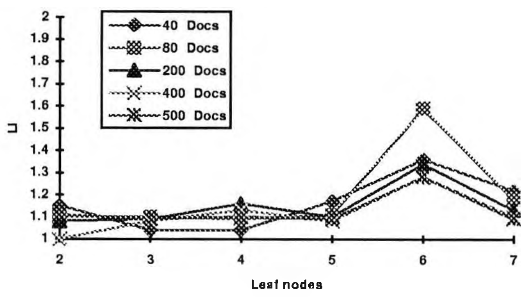


Fig 8-54. BASE1 [*TermId*]: index reorganisation load imbalance (postings only)

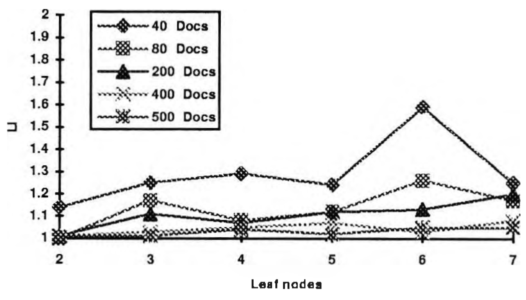


Fig 8-55. BASE1 [*DocId*]: index reorganisation load imbalance (position data)

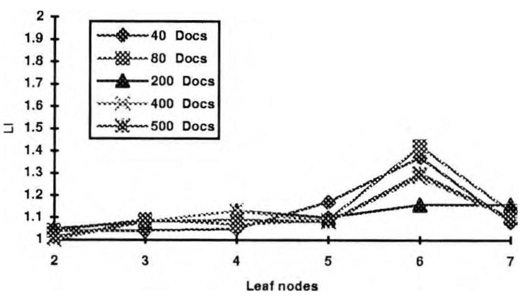


Fig 8-56. BASE1 [*TermId*]: index reorganisation load imbalance (postings only)

The imbalance figures for *DocId* partitioning show that initiating index reorganisations with a 40 document buffer does not yield good load balance. Increasing the leaf nodes set size also has a tendency to increase imbalance. The *TermId* partitioning method is generally more consistent, but imbalance on six leaf nodes for any buffer size is noticeably worse than for other leaf nodes. This is a failure of the distribution process which relies on a heuristic to distribute data to leaf nodes (see chapter 4, section 4.4.2). This has a direct and significant impact on speedup for *TermId* partitioning runs for 6 leaf nodes (see figs 8-49 and 8-50). Table 8-5 shows scalability results for index reorganisation.

Metric	**Collection**	**40 Docs**	**80 Docs**	**200 Docs**	**400 Docs**	**500 Docs**
Postings dealt with during index update	BASE1 BASE10	1.9 17.3	3.0 27.3	4.8 43.5	7.5 67.7	8.0 71.7
Postings Only (No Positions)						
Elapsed Time (secs)	BASE1 BASE10	5.49 43.8	6.90 56.0	10.2 80.8	16.9 121	17.3 106
Scalability on Elapsed Time (secs)	BASE10	1.25	1.23	1.26	1.40	1.64
Load Imbalance	BASE1 BASE10	1.65 1.49	1.30 1.20	1.12 1.10	1.10 1.07	1.09 1.06
Position Data						
Elapsed Time (secs)	BASE1 BASE10	13.8 135	17.2 170.7	24.3 231.0	39.7 336	43.6 351
Scalability on Elapsed Time (secs)	BASE10	1.02	1.01	1.05	1.18	1.24
Load Imbalance	BASE1 BASE10	1.68 1.60	1.32 1.30	1.12 1.11	1.14 1.09	1.09 1.03

Table 8-5. BASE1/BASE10 [*DocId*]: index update scalability on index reorganisation

One clear aspect of these results is that scalability is improved by having a much larger buffer size. The increase in scale occurs on both the elapsed time and on the total accumulated time for a reorganisation (see table A4-1 in appendix A4). The amount of data moved for the BASE10 collection is correspondingly larger but the proportion moved compared with BASE1 is much smaller being under half the collection for all runs. The total number of posting records for BASE10 is 223 million. Otherwise much the same trend is found in BASE10 as BASE1.

8.10.4 *Summary of Experimental Results*

In all aspects of transaction processing and index reorganisation, *DocId* partitioning is shown to be superior to *TermId* partitioning. For update transactions both methods are quick when data is added to the buffer, but *DocId* provides better transaction performance when an index reorganisation is being executed. Many update transactions meet our 100 millisecond requirement for elapsed times for document insertions. For transactions with both updates and queries, *DocId* is superior largely because of the performance improvement which is obtained with that method shown in chapter 7 on probabilistic search. The total number of records moved during index reorganisation is reduced with increasing numbers of leaf nodes when *DocId* partitioning is used. There is however an offset between the buffer size for incoming updates and increasing the leaf nodes set in order to reduce the amount of data moved. Overall our empirical results demonstrate that *DocId* partitioning is the preferred method for servicing the inverted file index maintenance techniques outlined in this chapter. One might question the viability of the method of index update, if queries are delayed beyond the 10 second response time recommended by Frakes (1992) or updates are delayed more than the 100 millisecond or 1 second requirement recommended in this chapter. This issue will be examined further in the next section.

8.11 SUMMARY AND CONCLUSION

Operators and the possible side effects found in the presence of an incorrect or non-existent CC mechanism have been identified. The three main CC mechanisms (lock, optimistic CC and Timestamp Ordering) are used to show how the side effects from these operations can be avoided, in particular the effect on document availability. Three semantics for document availability have been introduced, i.e. *Log-On*, *Late* and *Early*. *Late Availability* or *Log-On* semantics would be suitable for many IR systems, e.g. OPACS. We do not see the *Early Availability* semantics as being practicable at this point: *Late Availability* semantics is used in our practical experiments. Further work is needed at some stage to identify the complexity of the side effects found with mixed operations and how CC mechanisms are to be used in conjunction with inverted files.

The empirical results from this research show that in all aspects of both transaction processing and index reorganisation, the *DocId* partitioning method is far superior. Our synthetic model for the index update task is strong enough to distinguish between the partitioning methods (see chapter 5, section 5.6). Problems highlighted in our probabilistic search experiments (see chapter 7) impose severe restrictions on transaction processing when the *TermId* method is used, which are difficult to solve within our experimental context. These problems (most notably the sort aspect of search) had an impact on the relative difference between the two partitioning methods during transaction processing, a problem the synthetic model was not able to deal with very well. The synthetic model correctly predicted that the performance of transactions serviced on *TermId* indexes during an index reorganisation would deteriorate (although the empirical results were relatively worse because of the list size problem described in section 8.10.1, 2nd paragraph). The synthetic model also correctly predicted that transaction performance on *DocId* partitioning indexes would be constant after a given number of leaf nodes is reached in same situation. In index reorganisation when using *DocId* partitioning, the amount of data which needs to be moved reduces with increasing leaf node set size due to the qualities of the keyword set for each element of the leaf nodes set (the assumption made in the synthetic model was correct). Providing the same term block strategy was used, this effect will be a generic one. We have found evidence however, that *TermId* might be useful in a concurrent transaction processing context, and this would have to be the focus for any future research.

It may be the case that the methods outlined in this chapter for dealing with new documents may not be viable in a realistic situation: we could consider a scenario where the update rate was so high, buffer space would run out thereby crashing the system or cause a denial of service. Such a problem would occur when there are more updates being submitted to the system than it can handle, so that the time to re-organise a index with these new updates is greater than the actual time available on the system. There are limits to a method of storage such as inverted file which is designed for fast search and which is expensive to maintain: therefore for these high update applications some other method of transaction processing and storage method is required. Where our methods are not useable we would recommend the use of a two phase signature search (Cringean et al, 1991a; 1991b) or any of the vector processing methods (see chapter 2, section 2.4.7) which allow for cheap updates, but also allow for a high degree of parallelism.

Chapter 9

Passage Retrieval Results

9.1 INTRODUCTION

This chapter addresses the issue of applying parallel techniques to a large search space for passage retrieval that has been used by Okapi at TREC (Robertson et al, 1995). As explained in the introduction to this thesis, the algorithm is very computationally intensive: this method would require vast CPU resources and we apply parallel techniques to speed up the process. We describe the data and settings used in our main experiments in section 9.2. We discuss our main experimental results together in section 9.3. Experiments done at TREC-8 (MacFarlane et al, 2000a) are described in section 9.4. A conclusion is given in section 9.5.

9.2 DATA AND SETTINGS USED

The collections used in the experiments are BASE1 and BASE10. We use both *distributed* and *local* passage processing methods (see chapter 4, section 4.7.2): given the theoretical results from chapter 5, section 5.4, we describe results given on the *DocId* partitioning method only. For the *distributed* passage processing method we use the BASE1 collection only, creating indexes on 1 to 8 nodes, and initiate search experiments on all of those indexes. The BASE1 and BASE10 collections were used for the *local* passage processing method, running queries on 8 leaf nodes for BASE10 and 1 to 8 leaf nodes for BASE1. We use timings only from the AP3000, but ran experiments on the Alpha farm and used TREC evaluation techniques to confirm that the same document sets were retrieved for each topic.

The queries are based on topics 351 to 400 of the TREC-7 ad-hoc track: 50 queries in all (Voorhees and Harman, 1999). The terms were extracted from TREC-7 topic descriptions using an Okapi query generator. We used two types of queries: one based on *title only* (average number of terms per query is 2.46) and one based on the *whole topic* (average number of terms per query is 19.58). The *whole topic* query set has 51 queries, one extra being for VLC2 experiment initialisation (Hawking et al, 1999). Our timing methodology is as follows: for *title only* queries we declare the average of 10 runs, while we declared the average for 5 runs on *whole topic*.

The atom size used in our experiments is 5 sentences using an incremental step of one for atoms. For *distributed* passage processing we do passage retrieval on the top 1000. With *local* passage processing we vary the number of documents to which passage retrieval is applied on each processor: the reasoning for this is described in section 9.3. We select the top 20 of all runs for evaluation (relevance judgements for this data are restricted to this figure). The tuning constants used in the BM25 weighting function are as follows;

- 1.5 for K1 and 0.2 for B for all *title only* runs;
- K1=1.5 and B=0.4 for *whole topic* on BASE1 runs;
- K1=2.0 and B=0.6 for *whole topic* on BASE10 runs.

The settings for *title only* queries were found to be the best on the full 100GB VLC2/WT100g collection during preparatory experimentation for the TREC8 Web Track (see chapter 7, section 7.7). The constants for *whole topic* were found by further experimentation after our TREC8 experiments were complete.

9.3 EXPERIMENTAL RESULTS

Leaf nodes	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Type A	*1000* (1000)	*1000* (2000)	*1000* (3000)	*1000* (4000)	*1000* (5000)	*1000* (6000)	*1000* (7000)	*1000* (8000)
Type B (1000)	*var*	*var*	*var*	*var*	*var*	*var*	*var*	*var*
Type C	*1000* (1000)	*500* (1000)	*334* (1002)	*250* (1000)	*200* (1000)	*167* (1002)	*143* (1001)	*125* (1000)
Type D								
(504)	-	-	-	-	-	-	-	*63*
(256)	-	-	-	-	-	-	-	*32*
(128)	-	-	-	-	-	-	-	*16*

Key

var : A variable number of documents are processed on a leaf nodes.
1000 : The number of documents examined by a leaf nodes.
(1000) : A total of N documents is examined in that process configuration (all leaf nodes)

Table 9-1. Examples of documents processed per leaf nodes/total

We divide our experiments into four types: defined as types A to D: table 9-1 shows the number of documents processed per leaf nodes for each experiment type. Type A experiments apply the passage retrieval algorithm to the local top 1000 documents, therefore the maximum possible documents to be examined is 1000 times the number of leaf nodes; e.g. with 8 leaf nodes we examine 8000 documents. Type B experiments use the *distributed* passage processing method on the top 1000 documents for the whole database. In Type C experiments we examine a total of 1000 documents on a local basis. The number of documents examined by each leaf node in Type C experiments will vary according to the number of processes, e.g., with 2 leaf nodes we examine 500 document each etc: where 1000 documents exactly cannot be examined we inspect slightly more. For example using 3 leaf nodes we examined 334 documents each, giving a total of 1002 documents inspected. Type D is an extension of type C, but examining less than 1000 documents: we examined 504, 256 and 128 documents on 8 leaf nodes only.

To facilitate the discussion below (which is the examination of parallel runs) we declare sequential passage retrieval results used for comparison on the BASE1 collection. The average elapsed time in seconds is 1.33 for *title only* queries and 45.6 for *whole topic* queries. The figure for *title only* is acceptable and is within the scope of 10 second response times suggested by Frakes (1992). However *whole topic* response time is not acceptable within such a criterion: it does however demonstrate how computationally costly passage retrieval can be. Retrieval effectiveness is increased on all precision points when applying passage retrieval on the BASE1 collection using *title only* queries: precision at 20 is increased from 0.130 to 0.148 (an increase in performance of 13.8%). For all other runs improvements were found only on some of the precision points (see table 7-1 chapter 7, for effectiveness results on runs without passage processing). We use a number of different metrics to examine the performance of parallel passage retrieval methods: average elapsed time, system throughput (an estimate of queries processed per hour), load imbalance or LI, scalability, scaleup, speedup and efficiency (these metrics and our requirements of them are defined in chapter 3, section 3.4.1). We compare the elapsed times for VLC2 participants with our results in type A experiments only (these are the most expensive runs). We also examine other measures such as precision at 5 to 20 and passage retrieval statistics such as number of documents and passages processed during a run.

9.3.1 *Type A Experiment Results: Local Top1000 Documents*

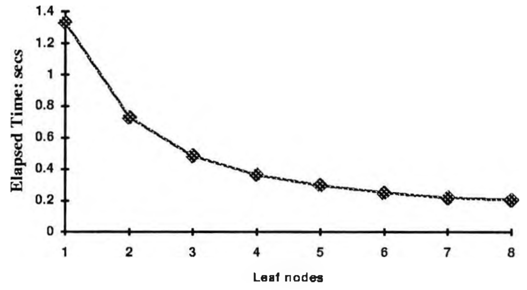


Fig 9-1. BASE1 [*title only*]: Type A retrieval efficiency, average elapsed time in seconds for passage retrieval

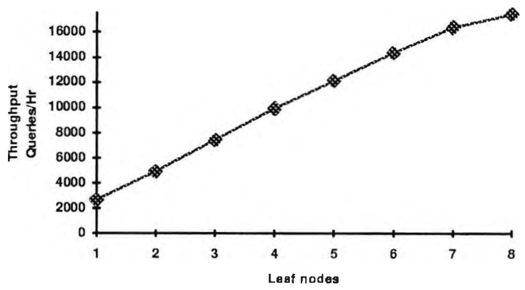


Fig 9-2. BASE1 [*title only*]: Type A retrieval efficiency, throughput (queries/Hour) for passage retrieval

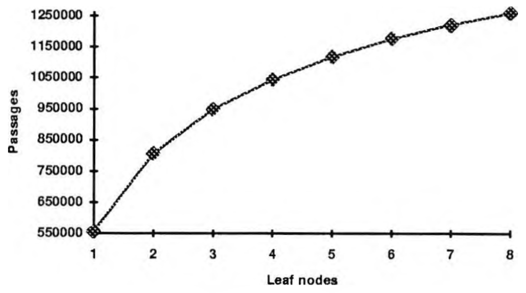


Fig 9-3. BASE1 [*title only*]: Type A retrieval efficiency, passages processed for passage retrieval

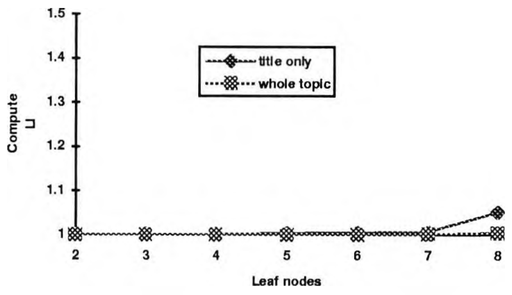


Fig 9-5. BASE1: Type A retrieval efficiency, load imbalance for passage retrieval

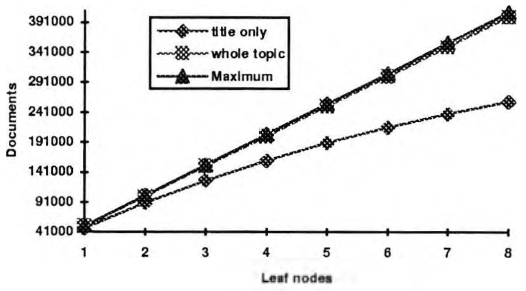


Fig 9-4. BASE1: Type A retrieval efficiency, documents processed for passage retrieval

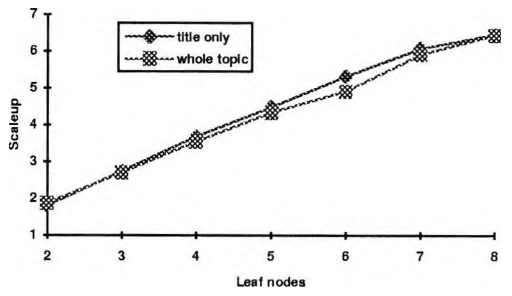


Fig 9-6. BASE1: Type A retrieval efficiency, scaleup for passage retrieval

The results on *title only* queries are very good. The elapsed time for every parallel run on BASE1 is under a second (see fig 9-1) and meets the 10 second requirement suggested by Frakes (1992). The run using 8 leaf nodes is faster than 8 out of the 11 runs submitted by other participants at VLC2 (Hawking et al, 1999). Overall the results show a time reduction for BASE1 parallel runs over the sequential run, with a linear increase in throughput (see fig 9-2). Scaleup is super linear and increases with more leaf nodes in the topology (see fig 9-6): although the number of extra documents examined actually decreases with more leaf nodes (see fig 9-4). This increase in performance with respect to scale can be explained in part by the number of passages processed as the leaf node set is increased (see fig 9-3). Passages processed from 2 to 7 leaf nodes grow at a very slow rate, from just over half a million passages at 2 leaf nodes to 1.2 million inspected at 8. The Load imbalance figures demonstrate that the workload is fairly distributed amongst leaf nodes (see fig 9-5). It should be noted that the slight drop in relative performance on 8 leaf nodes for *title only* queries is because client processes had to be mapped to one of the search leaf nodes and this affected the timings very slightly (this is apparent in the load imbalance figures - see fig 9-5).

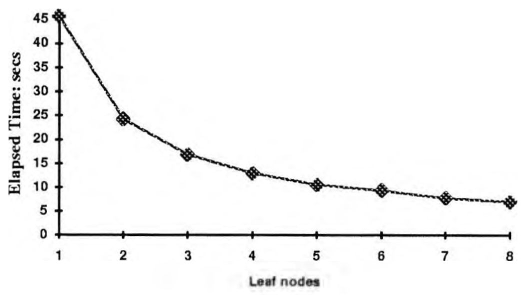


Fig 9-7. BASE1 [*whole topic*]: Type A retrieval efficiency, average elapsed time in seconds for passage retrieval

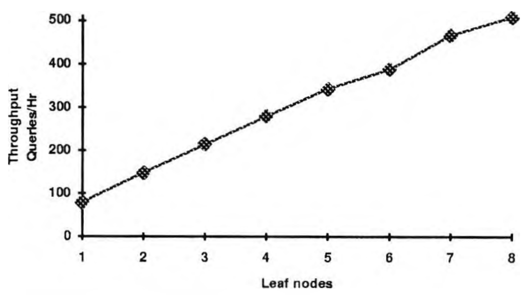


Fig 9-8. BASE1 [*whole topic*]: Type A retrieval efficiency, throughput for passage retrieval

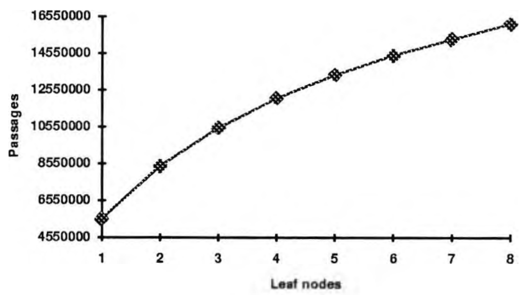


Fig 9-9. BASE1 [*whole topic*]: Type A retrieval efficiency, passages processed for passage retrieval

The elapsed time for *whole topic* queries on BASE1 (see fig 9-7) are not as good as title only, and runs to 5 leaf nodes do not meet the 10 second requirement for elapsed time suggest by Frakes (1992). Throughput is therefore much reduced compared to *title only* (see fig 9-8). The comparison with other VLC2 participants is not so good: the run at 8 leaf nodes only betters 2 out of the 11 submitted runs. While run times are much slower than *title only*, the other evidence found in those experiments are confirmed in these. Scaleup is super linear (see fig 9-6) which is surprising since the number of documents processed is very near the maximum (see fig 9-4). As with *title only* the number of passages processed reduces with increasing the number of leaf nodes, but *whole topic* queries shows slightly higher growth rate (see fig 9-9). The actual number of passages inspected is more with *whole topic* than with *title only* (see figs 9-3 and 9-9). Load imbalance is very small with *whole topic* queries (see fig 9-5).

Concerning retrieval effectiveness on *title only* it is found that no increase accrued for precision at 5,10 and 15 and it decreases on precision at 20 at various levels as number of leaf nodes is increased (see table 9-2). There are also slight variations in *whole topic* results (see table 9-3). However no decrease or increase found is statistically significant. We can state that examining the extra passages does not bring any benefit to any of the retrieval effectiveness measures used. It is also a clear indication that the ranking process is doing its job: the best documents useful for passage retrieval are contained in the top 1000 ranked documents.

Leaf nodes	**p@ 5**	**p@ 10**	**p@ 15**	**p@ 20**
1	0.268	0.186	0.161	0.148
2 to 4	0.268	0.186	0.161	0.147
5 to 8	0.268	0.186	0.161	0.146

Table 9-2. BASE1 [*title only*]: type A retrieval effectiveness results

Leaf nodes	**p@ 5**	**p@ 10**	**p@ 15**	**p@ 20**
1	0.196	0.160	0.140	0.126
2 to 8	0.200	0.162	0.141	0.127

Table 9-3. BASE1 [*whole topic*]: type A retrieval effectiveness results

Table 9-4 shows the comparison of BASE1 and BASE10 measures. Elapsed times in general are good apart from *whole topic* queries on BASE10: a minute or more response time for queries is not acceptable (Frakes, 1992). The average elapsed time for *title only* queries on BASE10 is better than half the runs of other VLC2 participants. The scalability derived from BASE1 to BASE10 is particularly good for *whole topic* queries which recorded 1.08 (a figure of 1.0 is linear scalability). Throughput on BASE1 is much better than BASE10 one would expect. All precision measures are better at BASE10 than BASE1: all record a reasonable increase. As with BASE1, there is little variation in BASE10 precision results for both types of queries.

MEASURE	**BASE1**		**BASE10**	
	title only	**whole topic**	**title only**	**whole topic**
Time (secs)	0.207	7.09	2.27	65.7
Throughput (Queries/Hr)	17,406	508	1,588	55
Scalability (1 to 10)	-	-	0.91	1.08
LI	1.052	1.005	1.015	1.003
p@ 5	0.268	0.200	0.320	0.348
p@ 10	0.186	0.162	0.304	0.310
p@ 15	0.161	0.141	0.275	0.275
p@ 20	0.146	0.127	0.251	0.249

Table 9-4. BASE1/BASE10: type A retrieval effectiveness and efficiency

In summary we state that while retrieval efficiency advantages are gained by using this type of parallelism (in spite of the extra passage data inspected), there is no clear gain in retrieval effectiveness. From the evidence given above the ranking process does its job well, therefore processing extra documents using our passage retrieval method is unnecessary when the BM25 weighting function is used.

9.3.2 *Type B Experiment Results: Distributed Top 1000 Documents*

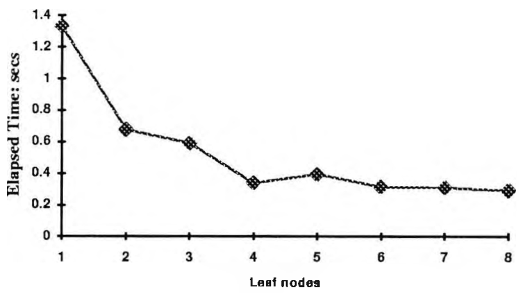


Fig 9-10. BASE1 [*title only*]: Type B retrieval efficiency, average elapsed time in seconds for passage retrieval

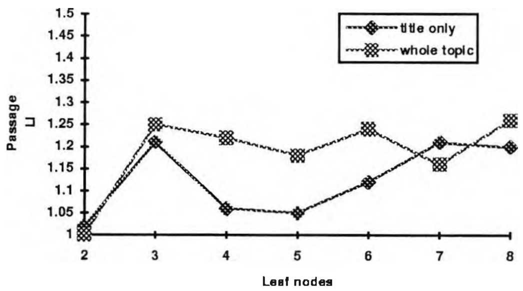


Fig 9-12. BASE1: Type B retrieval efficiency, load imbalance on passages processed for passage retrieval

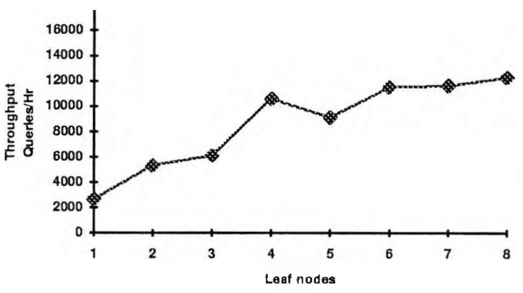


Fig 9-11. BASE1 [*title only*]: Type B retrieval efficiency, throughput (queries/hour) for passage retrieval

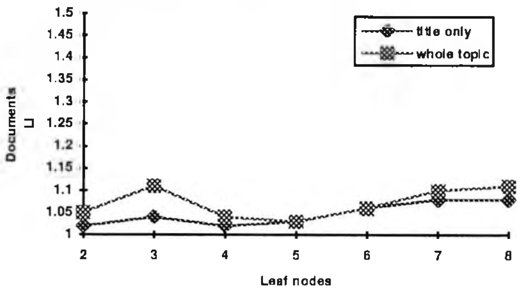


Fig 9-13. BASE1: Type B retrieval efficiency, imbalance on documents processed for passage retrieval

The results for the type B experiments on the BASE1 collection using *title only* queries can only be described as unpredictable and erratic. The throughput and parallel measurements show this effect (see figs 9-11 and 9-14 to 9-16). The results show that workload for small queries on passage processing may not be evenly distributed among leaf nodes. The evidence from imbalance on document and passages processed using *title only* queries, shows that imbalance on documents has little effect (see fig 9-13) but that there is an imbalance in processed passages which does have a negative effect (see fig 9-12). Communication overhead will account for some of the differences, but not wholly since communication increases linearly with leaf nodes. The rest of the imbalance must be because work on individual passages will vary greatly: the a(a-1)/2 cost estimate of the passage processing method used (see chapter 5, section 5.4) contributes significantly to imbalance. Some leaf nodes may get more expensive passages to compute than others. The only positive result from *title only* runs is that all parallel runs average under a second (see fig 9-10).

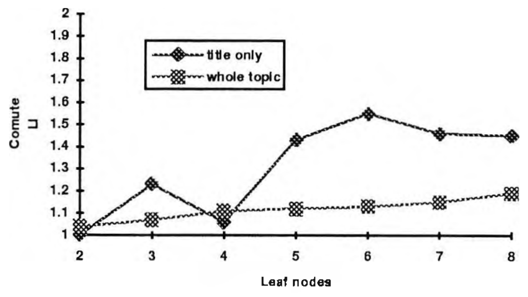


Fig 9-14. BASE1: Type B retrieval efficiency, computational load imbalance for passage retrieval

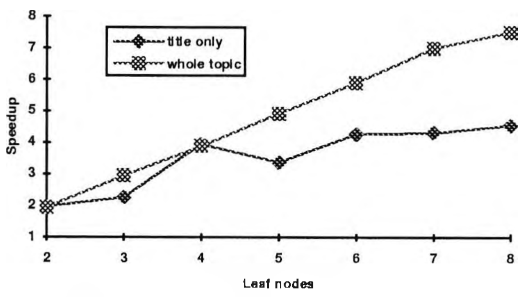


Fig 9-15. BASE1: Type B retrieval efficiency, speedup for passage retrieval

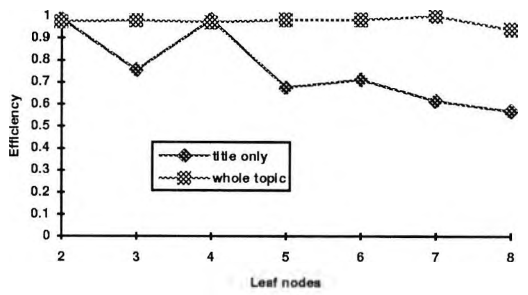


Fig 9-16. BASE1: Type B retrieval efficiency, parallel efficiency for passage retrieval

The performance of *whole topic* queries on the BASE1 collection is more predictable than *title only*. The parallel measures of speedup and efficiency show this with a near linear speedup (see fig 9-15) together with efficiency figures of just under 1 (see fig 9-16). The load imbalance in the *whole topic* experiments is also more predictable than *title only* with a linear deterioration in load balance as more leaf nodes are deployed (see fig 9-14). There is no correlation between the imbalance in processing costs and the imbalance of documents or passages processed. Elapsed time is poor with runs on 5 to 8 leaf nodes only meeting the 10 second criterion (see fig 9-17). Throughput is not very good, but this is only to be expected with a query set of the size of *whole topic* (see fig 9-18).

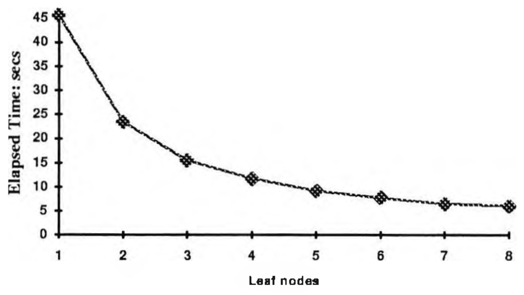


Fig 9-17. BASE1 [*whole topic*]: Type B retrieval efficiency, average elapsed time in seconds for passage retrieval

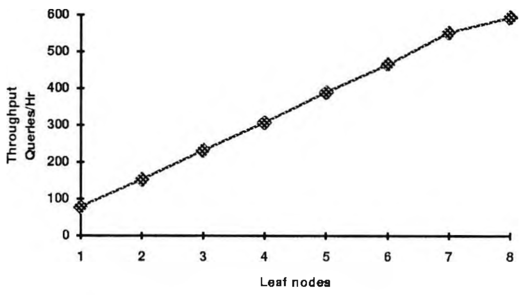


Fig 9-18. BASE1 [*whole topic*]: Type B retrieval efficiency, throughput (queries/ hour) for passage retrieval

The total numbers of documents and passages processed remain constant on all runs: 47,445 and 557,253 respectively for *title only* with *whole topic* values of 51,000 documents and just over 5 and a half million passages. Retrieval effectiveness results for the parallel algorithm are identical to the sequential algorithm for both types of query. It is hard to make a case for deploying this type of parallelism due to load imbalance problems inherent with the method.

9.3.3 *Type C Experiment Results: Local Top (1000/Leaf nodes) Documents*

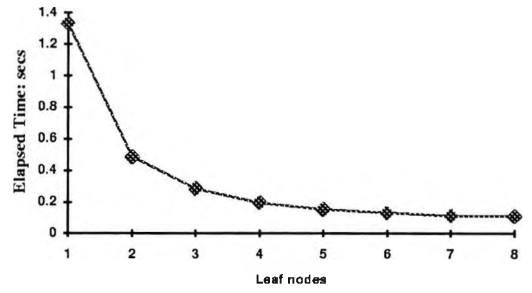


Fig 9-19. BASE1 [*title only*]: Type C retrieval efficiency, average elapsed time in seconds for passage retrieval

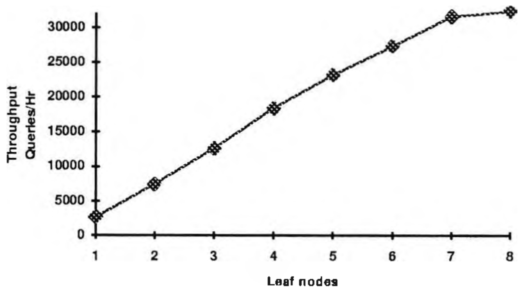


Fig 9-20. BASE1 [*title only*]: Type C retrieval efficiency, throughput (queries/ hour) for passage retrieval

The results gained in Type C experiments for *title only* queries can only be described as remarkable. The parallel measurements demonstrate this clearly with a super linear speedup recorded on all multiprocessor runs (see fig 9-24). As a result all efficiency results are greater than 1 with most figures greater than 1.5 (see fig 9-25). Load imbalance is very small with most figures very near 1 (see fig 9-23). All elapsed times are very good and under half a second (see fig 9-19). The scalability from BASE1 to BASE10 is very good, recording a figure of 1.32

(refer to table 9-6). The figures are all the more remarkable given that the numbers of documents and passages are virtually the same as in type B experiments (see figs 9-21 and 9-22). The method gains substantially from reducing communication overhead (no extra communication is needed as in type B). Why does a substantial speed advantage over sequential processing occur with this method of parallel passage retrieval? The passage retrieval algorithm examines a slightly different set of documents, which is less computationally intensive to examine than the set examined on the uniprocessor.

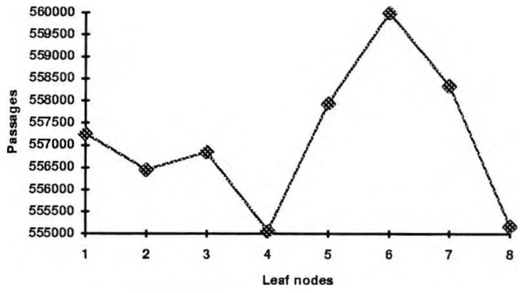


Fig 9-21. BASE1 [*title only*]: Type C retrieval efficiency, passages processed for passage retrieval

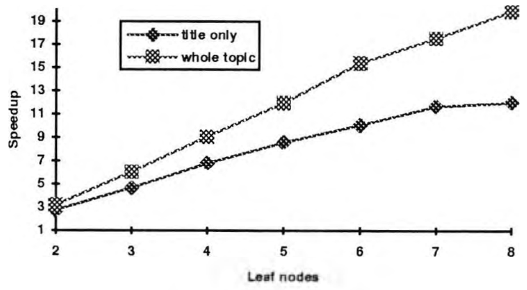


Fig 9-24. BASE1 [*title only*]: Type C retrieval efficiency, speedup for passage retrieval

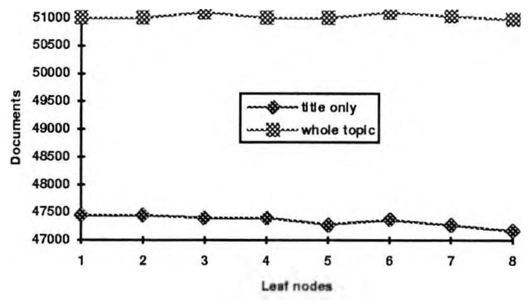


Fig 9-22. BASE1 [*title only*]: Type C retrieval efficiency, documents processed for passage retrieval

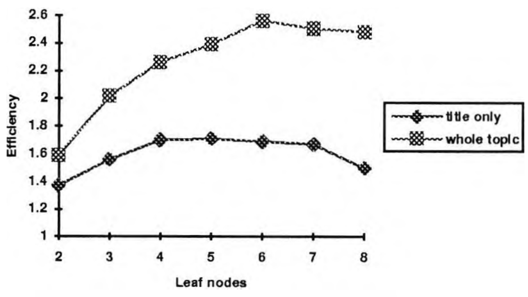


Fig 9-25. BASE1 [*title only*]: Type C retrieval efficiency, parallel efficiency for passage retrieval

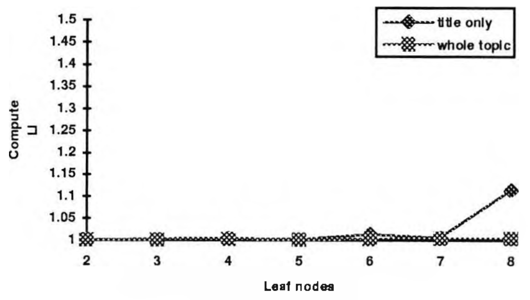


Fig 9-23. BASE1 [*title only*]: Type C retrieval efficiency, load imbalance for passage retrieval

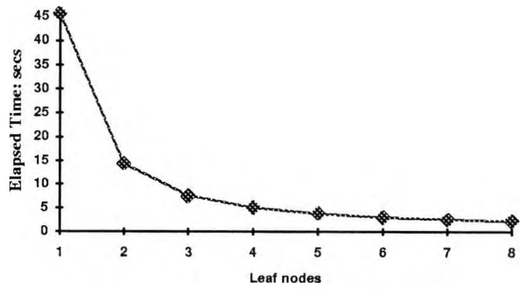


Fig 9-26. BASE1 [*whole topic*]: Type C retrieval efficiency, average elapsed time in seconds for passage retrieval

The query processing results are even more remarkable using *whole topic* queries than using the *title only* query set. A large part of the difference can be put down to the number of passages processed in type C experiments as compared with type B on *whole topic*: just over three million for the former compared with five and half million for the later (see fig 9-28). As with *title only* queries the numbers do not vary much on runs with differing numbers of leaf nodes. Response times for parallel runs are only unacceptable at 2 leaf nodes (see fig 9-26) and throughput is much improved in type C experiments compared to type B (see fig 9-27).

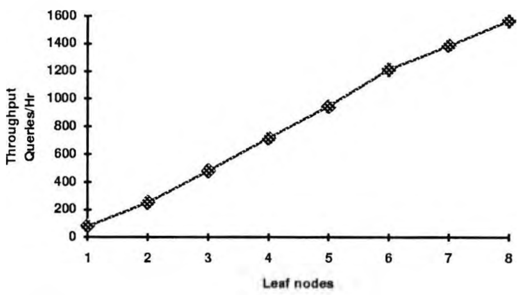


Fig 9-27. BASE1 [*whole topic*]: Type C retrieval efficiency, throughput (queries/ hour) for passage retrieval

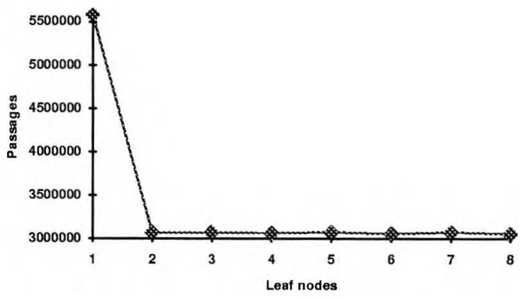


Fig 9-28. BASE1 [*whole topic*]: Type C retrieval efficiency, passages processed for passage retrieval

The retrieval effectiveness found in type B experiments are also found in type C experiments for both types of queries, e.g. 0.148 for precision at 20 for *title only*. Precision at 20 varies very slightly for *whole topic* queries, but the difference is not significant, e.g. values of 0.126/0.127 were recorded. From this we deduce that examining the top 1000 documents from the whole database is no different from examining 1000 documents by choosing locally on each leaf node. The evidence from this leads directly on to experimentation of type D (see the next section).

9.3.4 *Type D Experiment Results: Reduced Document Set*

As we have observed from the above, it does not seem to matter if we examine 1000 documents or more, retrieval effectiveness at lower precision points is not improved or harmed much by any of the methods applied. We decided to experiment with a smaller set of documents using the parallel passage retrieval algorithm on 8 leaf nodes with the BASE1 and BASE10 collections. This is done to see how the reduction affected the performance of the parallel algorithm. We chose 504, 256 and 128 being near a half, a quarter and eighth respectively of type B and C runs (we include no passages results in table 9-5 for comparison purposes).

Collection	Docs seen	Query Type	p@ 5	p@ 10	p@ 15	p@ 20
BASE1	1000	*title only* *whole topic*	0.268 0.200	0.186 0.162	0.161 0.141	0.148 0.127
	504	*title only* *whole topic*	0.268 0.200	0.186 0.162	0.161 0.141	0.147 0.127
	256	*title only* *whole topic*	0.268 0.200	0.184 0.162	0.160 0.141	0.146 0.127
	128	*title only* *whole topic*	0.268 0.196	0.184 0.160	0.159 0.140	0.149 0.126
	0	*title only* *whole topic*	0.244 0.188	0.178 0.172	0.149 0.145	0.130 0.128
BASE10	1000	*title only* *whole topic*	0.324 0.352	0.302 0.310	0.275 0.275	0.254 0.251
	504	*title only* *whole topic*	0.324 0.356	0.302 0.312	0.275 0.271	0.254 0.249
	256	*title only* *whole topic*	0.324 0.356	0.304 0.304	0.276 0.272	0.252 0.246
	128	*title only* *whole topic*	0.320 0.364	0.300 0.306	0.279 0.267	0.253 0.246
	0	*title only* *whole topic*	0.324 0.356	0.282 0.298	0.273 0.271	0.264 0.247

Table 9-5. BASE1/BASE10: type D retrieval effectiveness results

Table 9-5 confirms that we do not need to examine the full 1000 documents to achieve very nearly the same level of retrieval effectiveness at lower precision. There are some differences but they are very minor for both types of query. We can reduce the level of computation on the passage retrieval method and still obtain the extra retrieval effectiveness found when using that method. The experiments reinforce the assertion that the BM25 ranking function does its job well. This function in conjunction with passage retrieval applied to smaller document sets can improve retrieval effectiveness.

Measure	docs=1000		docs=504		docs=256		docs=128	
	BS1	BS10	BS1	BS10	BS1	BS10	BS1	BS10
Time (secs)	0.11	0.84	0.10	0.71	0.09	0.62	0.08	0.58
Throughput (Queries/Hr)	32k	4.2k	38k	5.1k	42k	5.8k	46k	6.2k
Scalability	-	1.32	-	1.35	-	1.37	-	1.34
LI	1.11	1.04	1.14	1.04	1.16	1.04	1.18	1.05

Table 9-6. BASE1/BASE10 [*title only*]: Type D retrieval efficiency results

From table 9-6 we can see that elapsed times for *title only* queries change only very slightly as the number of documents examined by the passage retrieval decreases. The times in seconds from term weighting searches (0.06 for BASE1, 0.54 for BASE10) show that while BASE10 shows linear reduction from 1000 to 128, BASE1 does not. This indicates that the size of collection combined with the term weighting scheme are significant factors that affect passage retrieval performance. BASE10 being a much larger database will pick documents that are roughly the same in compute terms with respect to passages and they are ranked in the top set by the term weighting scheme. BASE1 ranks more passage intensive documents higher up the rank which require extra computation: further evidence that the ranking process is doing its job. Other measures such as scalability and LI are good.

The figures for *whole topic* queries show more improvement than *title only* queries with respect to average query processing time (see table 9-7). The timings from term weighting searches (0.77 for BASE1, 6.45 for BASE10) show that the time reduction is very near linear. Load balance for all runs is very good. The scalability is excellent for all BASE10 runs, and confirms the effect of collection size and term weighting function on performance.

Measure	**docs=1000**		**docs=504**		**docs=256**		**docs=128**	
	BS1	**BS10**	**BS1**	**BS10**	**BS1**	**BS10**	**BS1**	**BS10**
Time (secs)	2.29	16.3	1.64	10.4	1.28	8.47	1.09	7.50
Throughput (Queries/Hr)	1.5k	221	2.2k	346	2.8k	425	3.3k	480
Scalability	-	1.41	-	1.58	-	1.52	-	1.45
LI	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01

Table 9-7. BASE1/BASE10 [*whole topic*]: Type D retrieval efficiency results

9.4 TREC8 AD-HOC TRACK EXPERIMENTS

TRACK RUN-ID	**QP TYPE**	**QUERY TYPE**	**COMMENTS**
plt8ah1	PASSAGE	Title Only	2 Timing runs
plt8ah2	PASSAGE	Title/Description	2 Timing runs
plt8ah3	PASSAGE	Title/Descr./Narr.	2 Timing runs
plt8ah4	Term W.	Title Only	4 Timing runs.
plt8ah5	Term W.	Title/Description	4 Timing runs.
plt8ah6	Term W.	Title/Descr./Narr.	4 Timing runs.

Table 9-8. Details of TREC8 Ad-Hoc track runs

The purpose of the TREC8 Ad-Hoc experiments is twofold. Firstly we examine the issue of probabilistic search with no passages (referred to simply as term weighting in the rest of this chapter) versus passage retrieval search (see section 9.4.1). Secondly we wanted to confirm our experience with type A experiments described in section 9.3.1, that examining more of the search space in passage retrieval does not bring any benefit (see section 9.4.2). We used the Ad-Hoc data for this as a fuller set of relevance judgments is available compared with the full 100 Gbyte web collection or its baselines. We used the "Cambridge Cluster" for the experiments described here plus a single Pentium machine for comparison purposes. We submitted five runs and recorded times for the two types of search on queries derived from topics 401-450. For passage retrieval we did one parallel run and one uniprocessor run. For term weighting we did the same but applied them on indexes with postings only data (ordinary term weighting using BM25 does not use position information statistics). We prepared *title only*, *title/description* and *title/description/narrative* queries for each topic from Okapi generated queries. The run identifiers together with their query processing type and query type are given in table 9-8.

The tuning constants for *title only* are: **K1**=2.5 and **B**=0.9. For *title/description* we used a **K1** value of 2.5 and a **B** value of 0.6. A value of 4.0 for **K1** and 0.7 for **B** is used for *title/description/narrative* queries. The choice of this tuning constant data is based on experiments described in MacFarlane et al (2000a). The average length of the queries is: 2.42 for *title only*, 9.88 for *title/description* and 24.74 for *title/description/narrative*.

9.4.1 *Retrieval Efficiency*

RUN-ID	SEQ. TIME	PAR. TIME	SCALEUP	LI	EXTRA COST (SEQ)	EXTRA COST (PAR)
plt8ah1	0.826	0.357	2.31	1.06	2.06	4.19
plt8ah2	4.13	2.55	1.62	1.002	2.23	10.07
plt8ah3	11.6	8.21	1.41	1.003	2.34	18.52

Table 9-9. TREC8 Ad-Hoc retrieval efficiency results (passage retrieval)

RUN-ID	SEQ. TIME	PAR. TIME	SPEEDUP	EFFIC-IENCY	LI
plt8ah4	0.401	0.085	4.71	0.295	1.41
plt8ah5	1.85	0.253	7.31	0.457	1.35
plt8ah6	4.95	0.444	11.1	0.697	1.05

Table 9-10. TREC8 Ad-Hoc retrieval efficiency results (term weighting search)

The retrieval efficiency results for our TREC-8 Ad-Hoc runs using passage retrieval are declared in table 9-9 (we include term weighting results in table 9-10 for interest). The response times for all 6 parallel runs were good with all average query processing times under 10 seconds: 4 of them were under a second. The single processor runs registered good times with a maximum of 11.6 seconds for *title/description/narrative*, while 5 of the 6 runs had an average query processing time of under 5 seconds. With passage retrieval the parallel version examines 16 times more documents (16,000 as against 1,000) than the sequential version. All the passage retrieval runs managed to reduce the average query processing time and still examine the extra data: the best example is *title only* that took 0.83 seconds on average (sequential run) as against 0.36 on average for the parallel run. We record better than linear scaleups on all parallel runs which apply passage retrieval to query processing.

With respect to the extra costs of passage processing there is a marked change comparing search times on the parallel processing runs with much larger times for passage retrieval over ordinary term weighting search: the worst being a factor of 18.52 when using *title/description/narrative* queries. There is a clear correlation between these extra costs and query size (see table 9-9). It should be noted that the figures declared in this section and used for comparison are optimistic given that each leaf node in the "Cambridge Cluster" has three times the memory of the uniprocessor Pentium used.

9.4.2 *Retrieval Effectiveness*

Table 9-11 shows our Ad-Hoc results for TREC8. We restrict our discussion to average precision. We include in the table the results for submitted runs, and the best values found by varying the tuning constants **K1** and **B**.

TRACK RUN-ID	AVEP	BEST AVEP (CONST VALS)
plt8ah1	0.212	0.238 (K1=1.0,B=0.3)
plt8ah2	0.190	0.189 (K1=1.5,B=0.6)
plt8ah3	0.165	0.161 (K1=1.5,B=0.8)
plt8ah4	0.181	0.234 (K1=1.0,B=0.3)
plt8ah5	0.180	0.190 (K1=1.5,B=0.6)
plt8ah6	0.150	0.157 (K1=1.5,B=0.8)

Table 9-11. Average precision results for TREC8 Ad-Hoc runs

The original results submitted were on the low side. Results on long queries are not particularly good for any runs. The passage retrieval *title only* revised run (plt8ah1) produced results in which 24 out of the 50 topics were better than the median. This figure is reduced to 22 out of 50 for the revised term weighting run (plt8ah4). In both of these runs we record an average precision for topic 431 which is better than the best Ad-Hoc run: 0.558 compared to 0.526.

With respect to retrieval effectiveness gain of passage retrieval over term weighting we found that there were slight improvements for submitted runs, but for the best term weighing constant values there is little improvement and for *title/description* queries we actually recorded a slight reduction in average precision. It seems likely from this that the best tuning constants in term weighting search may not be the best pair when applied to query processing using passage retrieval: all tuning constants applied to Ad-Hoc runs were gathered on term weighting runs. The results here confirm the ones give in section 9.4.1 that examining extra search space by processing more passages does not bring much benefit if any.

9.5 SUMMARY AND CONCLUSION

The retrieval efficiency results given in this chapter are in the main very good indeed depending on the type of passage processing used or query set applied. With *local* passage processing the performance improvement using parallelism is very good on both types of query, particularly for type C experiments. For *distributed* passage processing whole topic shows benefit by using parallelism, but the results for *title only* queries is disappointing on any type of parallel measurement used. The elapsed time for any run on *title only* queries were well under the 10 seconds recommended by Frakes (1992), while the use of parallelism on *whole topic* queries does yield acceptable run times when using a larger processor set. Our synthetic model correctly predicted that the *local* passage processing method would outperform the *distributed* version, but is unable to predict the relative difference due to the erratic the nature of the performance in the latter (see chapter 5, section 5.4.2) on short queries.

With respect to retrieval effectiveness, the passage retrieval algorithms do bring benefits on web data over ordinary term weighting but such is not guaranteed. However it is clear from the above that the algorithm only needs to be applied to a subset of the top ranked documents to gain effectiveness. Clearly the ranking process using BM25 does its job very well. However gains are collection and query dependent. For example in our web data experiments, *title only* queries on the BASE1 database yield an increase of 13.8% from 0.130 to 0.148 in precision at 20 when passage retrieval is applied. A reduction in effectiveness on some precision points is recorded when the same queries are applied to the BASE10 collection: any increase found is not significant. *Whole topic* queries do not do as well as *title only*: this is a problem shared with probabilistic search (see chapter 7, section 7.3) and one which merits further investigation.

While efficiency improvement is significant using parallelism on passage retrieval, many of the experiments described above do not bring any benefits with respect to effectiveness over the sequential passage retrieval algorithm. A hypothesis that asserts that examining more of the search space in passage retrieval is effective is not supported by the results given in this chapter - indeed, the evidence is against it. This is not to suggest that applying parallelism to the passage retrieval does not work, but the choice will depend on whether to use parallelism with term weighting or not. We can apply the passage retrieval algorithm to fewer documents, at a computational cost slightly higher than that term weighting to obtain better retrieval effectiveness. The implication is that parallelism is not worth applying to passage retrieval on a reduced set of documents, if it is not worth applying the strategy to term weighting.

Chapter 10

Routing/Filtering Results

10.1 INTRODUCTION

This chapter describes results using parallel techniques that can be applied to the routing/filtering query optimisation techniques used by Okapi at TREC. The routing task we are considering is the situation where a number of relevance judgements have been accumulated and we want to derive the best possible search formulation for future documents. The filtering task is an extension of this, but a threshold for documents is applied, i.e. a binary yes/no decision is made on one document at a time as to whether it will be presented to the user. We describe the data and settings used for both the Ziff-Davis and TREC8 experiments. We then describe results gained on the Ziff-Davis test set, and then results gained on the TREC8 test set. The data collected for TREC8 was originally published in MacFarlane et al (2000a) but is expanded here. A conclusion is given at the end.

10.2. DATA AND SETTINGS USED

The databases used for the experiments were the Ziff-Davis collection and the TREC8 filtering track collection. Given the theoretical results given in chapter 5, section 5.5, we report results on two distribution strategies: *on-the-fly* and *replication* (for a description of these methods refer to chapter 1, section 1.4). The following term selection strategy was used on both data sets. Evaluations were done on the top 1000 ranked documents using the TREC average precision scoring method. The optimisation process was done on 300 terms per query. The document scores were varied by factors of 0.67 and 1.5 in the reweighting process: each term was tried at the unaltered weight, lower weight then higher weight; the first successful factor being chosen. The utility functions used are those described in chapter 3, section 3.4.2. A number of different stopping criteria were used: reaching a maximum time for a routing session on a topic, reaching a maximum number of iterations, discovering that no single term increases the score and discovering that a score is within a given range of increase. In most cases it was found that term selection stopped when no single term increased the score: this is consistent with experience found in Okapi at TREC. The threshold for gain on evaluations was 0.001% increase in score: the range for the last stopping criterion being between 0.00001% and 0.001%. A binary decision on the acceptance of terms is used based on these thresholds. Terms that did not increase the score were skipped after 4 evaluation failures and only re-examined after more than ten inner iterations had passed. These rules were used to ensure consistency with Okapi at TREC.

The timing metrics we use to measure retrieval efficiency are as follows (for formal definitions and our requirements for most of them see chapter 3, section 3.4.1). For each run we declare the average elapsed time for term selection in seconds for the topics. We define selection efficiency as an improvement in average elapsed time on the term selection algorithms by using parallelism. We use the standard parallel measures: speedup and parallel efficiency. We record the average number of iterations needed to optimise queries and the average overhead per iteration in seconds. The overheads at the synchronisation point are declared in seconds, averaged over iterations. For *on-the-fly* we declared the average load time for a topic. We examine retrieval efficiency for *replication* in greater depth: we collected more detailed information when doing these experiments. In order to measure the level of load imbalance in *replication*, we use the LI metric. The number of evaluations per second (throughput) is also declared for *replication* results. For the Sun Network version of the routing we present results from 1,2,4,6,8 and 10 slave nodes. We present results for the AP1000 version from 10 slave nodes up to 100 slave nodes at intervals of 10 as many more slave nodes were available. The distribution method on these experiments was *on-the-fly*. On the AP3000 we present runs on 2 to 7 slave nodes, while for the "Cambridge Cluster" we declare results on 1-15 slave nodes. The distribution method on these experiments was *replication*. The results for the uniprocessor on the network experiments are estimates based on a 1 *master*, 1 slave node configuration with the distribution time removed.

We use the following evaluation techniques declared in chapter 3, section 3.4.2. For the routing task we use recall and average precision metrics. For filtering on Ziff-Davis data we use the utility functions defined for the TREC-4 and TREC-5 tasks (Lewis, 1996 and 1997a). For filtering on TREC8 data we use utility functions defined for that task by Hull and Robertson (2000).

10.3. ZIFF-DAVIS EXPERIMENTS

We describe the experimental results both in terms of retrieval effectiveness and efficiency. We describe the efficiency results for both strategies, but effectiveness results are only discussed for *on-the-fly*. We discuss retrieval efficiency results for *replication* in detail, but only give an outline for *on-the-fly*: the reason for this will become clear. Note that ADD in the diagrams specifies the *add only* operation, A/R specifies *add/remove*, while RW refers to *add reweight* operation

10.3.1 Retrieval Efficiency Results: Replication

10.3.1.1 Elapsed Time for Term Selection

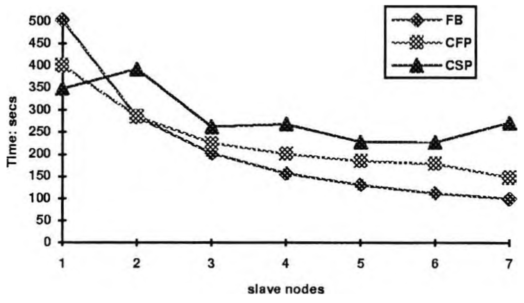


Fig 10-1. ZIFF-DAVIS [*Replication*]: *add only* average term selection elapsed time in seconds (AP3000)

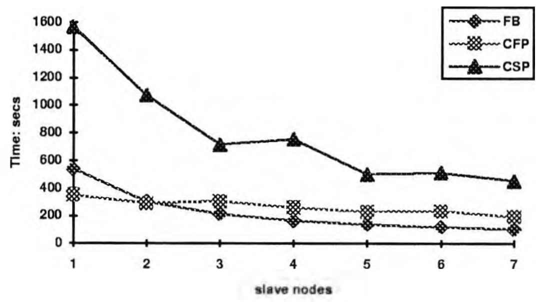


Fig 10-2. ZIFF-DAVIS [*Replication*]: *add remove* average term selection elapsed time in seconds (AP3000)

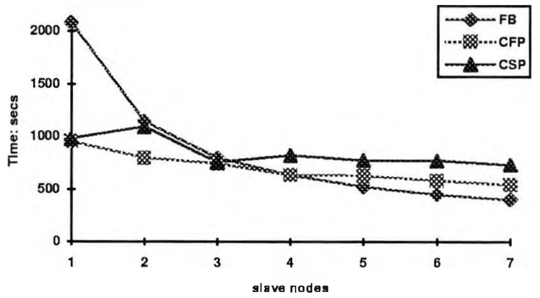


Fig 10-3. ZIFF-DAVIS [*Replication*]: *add reweight* average term selection elapsed time in seconds (AP3000)

From figs 10-1 to 10-3 it can be seen how expensive the application of the term selection algorithms can be, with average elapsed time running into hundreds of seconds in some cases thousands of seconds. Parallelism has different effects on individual term selection methods which can be either beneficial or detrimental: more details are given in the sections below. The FB algorithm is the term selection method which benefits most from the application of multiple slave nodes, showing a linear time reduction on all node set sizes. FB also outperforms the other term selection algorithms using larger node sets (this can be seen from all data presented in figs 10-1 to 10-3). Linear time reductions are also found with most parallel runs on CFP using any operation (this is most noticeable with *add only* operation - see fig 10-1). With regard to CSP using all studied operations, elapsed times do not follow any trend and vary unpredictably with slave node set size (particularly using *add only* operation - see fig 10-1). The most expensive operation in the majority of cases is *add reweight*: for example FB run times are roughly four times as slow on *add reweight* as the other operations. It is generally more expensive to use the *add/remove* operation compared with *add only* particularly with the Find Best algorithm.

10.3.1.2 Evaluation Throughput and Workload

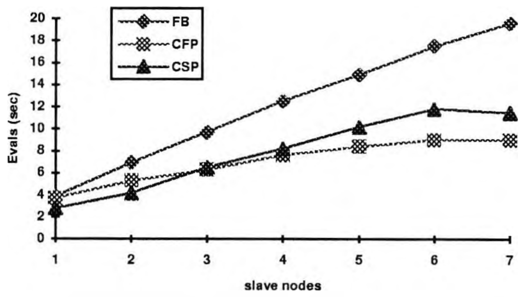


Fig 10-4. ZIFF-DAVIS [*Replication*]: *add only* throughput in evaluations per second (AP3000)

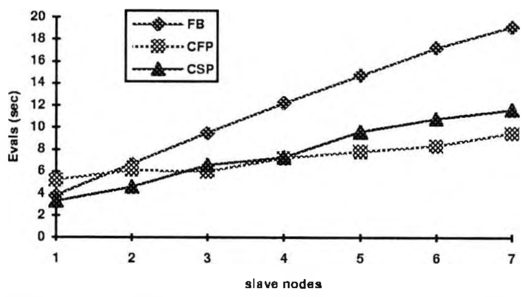


Fig 10-5. ZIFF-DAVIS [*Replication*]: *add remove* throughput in evaluations per second (AP3000)

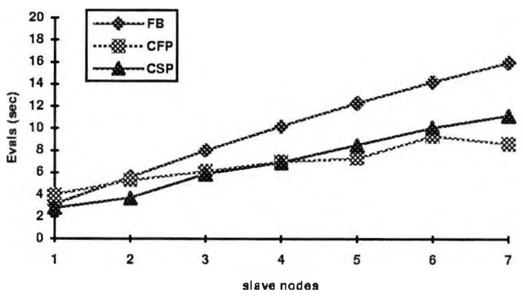


Fig 10-6. ZIFF-DAVIS [*Replication*]: *add reweight* throughput in evaluations per second (AP3000)

The throughput in evaluations per second for experiments on *replication* is shown in figs 10-4 to 10-6. The FB algorithm gets the most benefit with respect to throughput using parallelism. In all operations with this term selection method, a linear increase in throughput is recorded particularly for *add only* (see fig 10-4). The number of evaluations for any operation on FB remain constant for any parallel machine size. From CFP results it clear that gains in throughput are increasingly hard to come by: although the number of evaluations per topic does increase with size of parallelism. For example CFP with *add reweight* increases the total evaluations from 3787 on 1 slave node to 5434 on 7 slave nodes: the same trend is found with other operations. The increase in workload is an important part of the overall loss of parallel efficiency for the CFP algorithm (see below). Throughput on the CSP algorithm increases linearly with more parallelism, but does not match FB figures. With respect to CSP, a linear increase in evaluations per topic was found with *add only* and *add reweight* operations: the figure for *add reweight* was particularly significant with an increase of 2766 evaluations per topic at 1 slave node to 8234 at 7 slave nodes. These figures demonstrate that we are able to search more of the space, while still improving throughput and elapsed time performance. The number of evaluations per topic on CSP *add remove* varies with the parallel machine size.

10.3.1.3 Load Imbalance

The load imbalance metric used here is defined in chapter 3, sub-section 3.4.1.8. The imbalance for term selection is low and does not reach a point where load balance is a significant problem for the algorithms: for example an LI of 2.0 would mean halving the effective speed of the machine and the LI figures in figs 10-7 to 10-9 are nowhere near that level. However, general trend for load imbalance for most experiments is upwards. The exception is CSP with *add remove* operation which shows a reduced level of load balance over all runs. There is a clear increase in load imbalance as the number of slave nodes is increased, which demonstrates the need for some form of load balancing technique if many more slave nodes were to be used in optimising on a test set of this size. This imbalance contributes in part to the overall loss in term selection efficiency found with various measures.

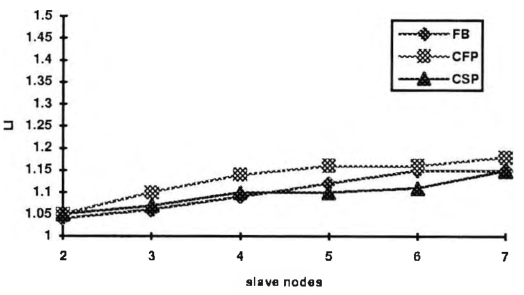


Fig 10-7. ZIFF-DAVIS [*Replication*]: *add only* load imbalance for term selection (AP3000)

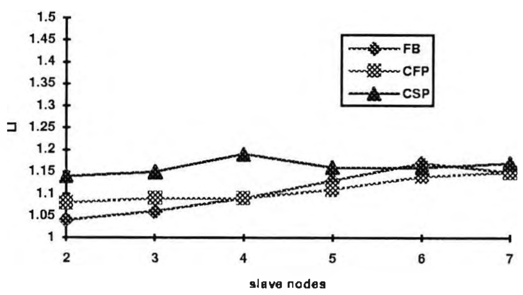


Fig 10-8. ZIFF-DAVIS [*Replication*]: *add remove* load imbalance for term selection (AP3000)

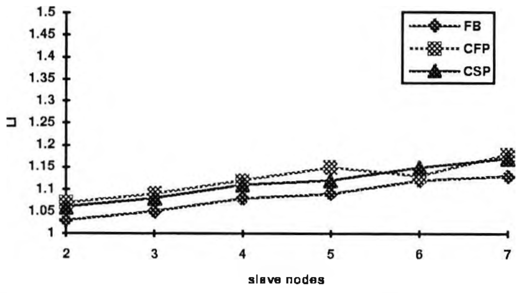


Fig 10-9. ZIFF-DAVIS [*Replication*]: *add reweight* load imbalance for term selection (AP3000)

10.3.1.4 Outer Iterations and Overheads

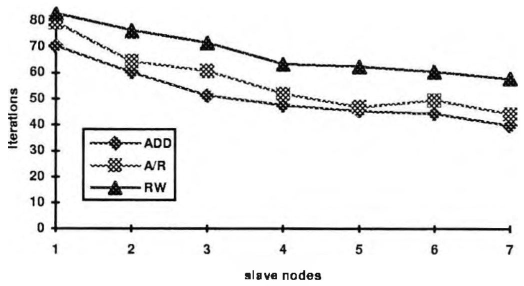


Fig 10-10. ZIFF-DAVIS [*Replication*]: Choose First Positive average outer iterations for term selection (AP3000)

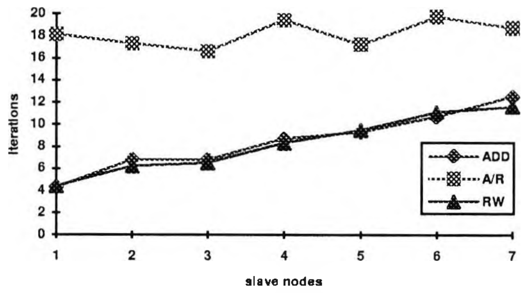


Fig 10-11. ZIFF-DAVIS [*Replication*]: Choose Some Positive average outer iterations for term selection (AP3000)

FIND BEST (FB)		
ADD	*A/R*	*RW*
20.3	20.2	29.7

Fig 10-12. ZIFF-DAVIS [*Replication*]: Find Best average outer iterations for term selection (AP3000)

The averages for outer iterations are shown in figs 10-10 to 10-12. The average number of iterations need to optimise topics for FB was found to be constant for any given operation, but variations did occur between them (see fig 10-12). The *add remove* operation average iterations was slightly less than *add only*, therefore selection on *add remove* is more expensive in time per iteration. The figure for *add reweight* is nearly 50% larger, which explains in part why using this operation with the FB method is so costly.

The Choose First term policy in the CFP algorithm increases the number of iterations needed to select terms for any operation dramatically compared with the others (see fig 10-10). However the number of iterations reduces with an increasing number of slave nodes used in a parallel machine. This factor accounts for some of the time reduction with the CFP algorithm. The gain from the application of parallelism is restricted: however the effect parallelism has on the term selection technique is significant. The operator applied also has an effect on the progress of CFP: there is a transitive relation on the number of iterations needed, i.e. *add only* iterations < *add remove* iterations, *add only* iterations < *add reweight* iterations and *add remove* iterations < *add reweight* iterations. Therefore the term operator has a direct effect on the number of terms chosen when CFP is used.

With CSP the number of iterations is much reduced compared with the other two algorithms and individual iterations are much costlier with this method (see fig 10-15 below).

There is an increase in the number of iterations for CSP *add only* and *add reweight* which has a direct causal effect on the total increase in the number of evaluations and explains any loss in term selection efficiency (see fig 10-11). Unlike the other two operations, in CSP with *add remove* the number of iterations per topic varies erratically as do the number of evaluations per topic on differing numbers of slave nodes (see sub-section 10.3.1.2).

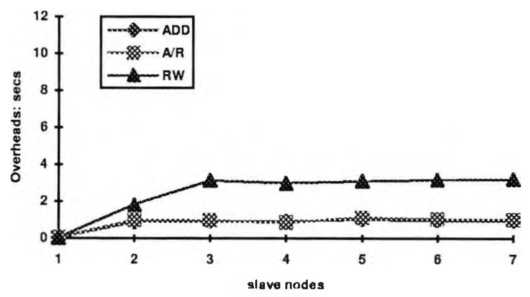


Fig 10-13. ZIFF-DAVIS [*Replication*]: Find Best average overheads in seconds for term selection (AP3000)

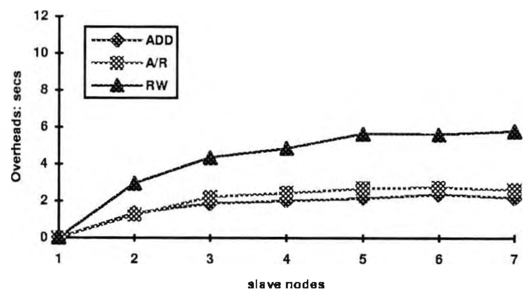


Fig 10-14. ZIFF-DAVIS [*Replication*]: Choose First Positive average overheads in seconds for term selection (AP3000)

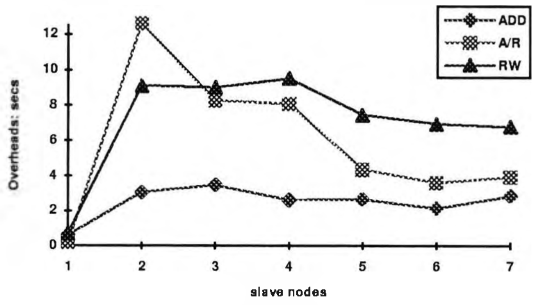


Fig 10-15. ZIFF-DAVIS [*Replication*]: Choose Some Positive average overheads in seconds for term selection (AP3000)

The overheads at the synchronisation point are shown in figs 10.13 to 10.15. Recall that the synchronisation point is where the stopping criterion is checked, the best term(s) is found and results communicated to all slave nodes (see chapter 4, sub-section 4.8.3.4). FB *add only* and *add remove* overheads while remaining fairly constant become increasingly important as processing time is reduced (see fig 10-13). FB *add reweight* overheads are much higher than the other two operations: this is generally true of the other operations as well. With CFP there is an increase in overheads when using more slave nodes, but this increase gradually tails off for any operation (see fig 10-14). However, given the level of iteration needed for that algorithm the overheads are very significant. With CSP overheads are less important for any operation, and some reduction in overheads is recorded for larger slave node sets on *add remove* and *add reweight* operations (see fig 10-15). Because of the amount of data needed to be exchanged at the synchronisation point for CSP, costs are much heavier in real terms per iteration than the other two algorithms. Overall this data shows that the synchronisation point for the algorithms studied is expensive and should be kept to a minimum.

10.3.1.5 Speedup and Parallel Efficiency

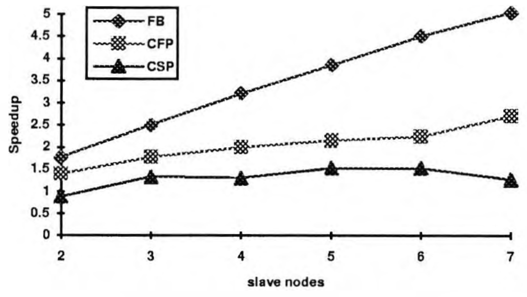


Fig 10-16. ZIFF-DAVIS [*Replication*]: *add only* operation speedup for term selection (AP3000)

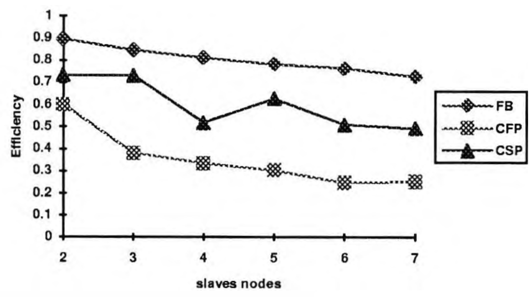


Fig 10-19. ZIFF-DAVIS [*Replication*]: *add remove* operation parallel efficiency for term selection (AP3000)

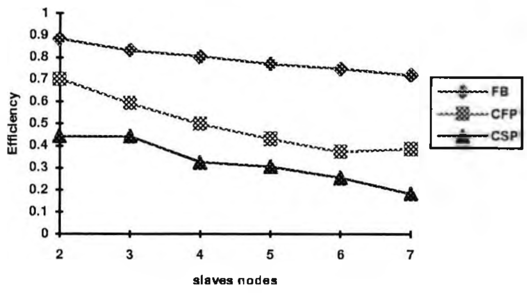


Fig 10-17. ZIFF-DAVIS [*Replication*]: *add only* parallel efficiency for term selection (AP3000)

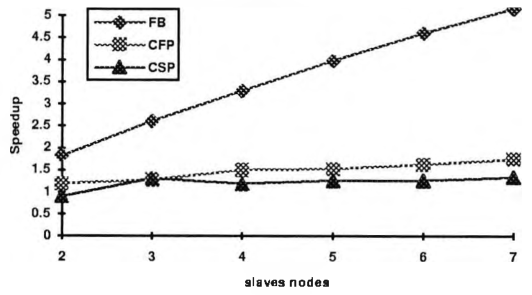


Fig 10-20. ZIFF-DAVIS [*Replication*]: *add reweight* operation speedup for term selection (AP3000)

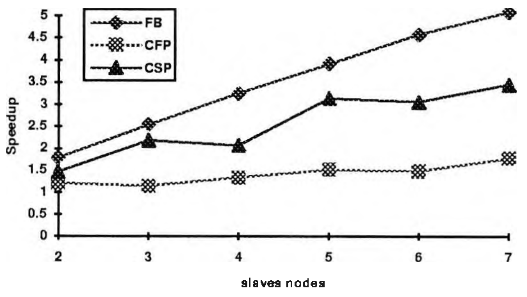


Fig 10-18. ZIFF-DAVIS [*Replication*]: *add remove* operation speedup for term selection (AP3000)

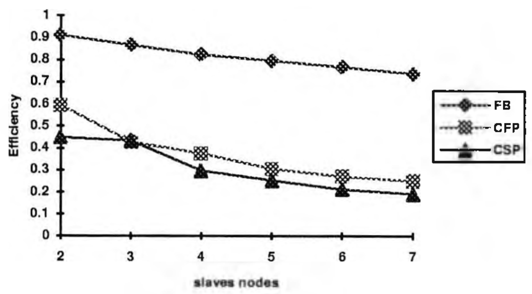


Fig 10-21. ZIFF-DAVIS [*Replication*]: *add reweight* operation parallel efficiency for term selection (AP3000)

The speedup and efficiency figures are shown in figs 10-16 to 10-21: these metrics are declared in chapter 3, sub-sections 3.4.1.4 and 3.4.1.5. In terms of speedup and parallel efficiency, the FB method shows improvement on all levels of parallelism investigated. Speedup is near linear at 7 slave nodes with parallel efficiency above the 70% mark for any operation. However the speedup and parallel efficiency for CFP is very poor for all three term operations. In most cases a speedup of less than two is registered and a number of factors are responsible for poor parallel performance. The increase in evaluations with more slave nodes is a significant factor as well as the overhead at the synchronisation point together with load imbalance (see above). Much the same can be said for CSP, apart from *add remove* operation which does actually show some level of speedup. However, overheads are a much less significant factor for CSP while the increase in evaluations play a more important part. Slowdown for CSP on *add only* and *add reweight* is recorded for 2 slave nodes. It could be argued that using speedup and parallel efficiency to measure the parallel performance of the CSP algorithm is unfair as the parallelism itself imposes an extra workload for the method. However demonstrating that some parallel performance improvement is available while still being able to examine some of the search space is, we believe, worthwhile.

10.3.1.6 Restriction of Outer iterations on CAP/CSP

In order to discover the effect of restricting the number of outer iterations on CSP to the level of CAP we ran further experiments on that term selection algorithm. We measure the performance of the CSP on the same iterative level as CAP: this is done by stopping CSP term selection early when it has reached the same number of iterations as CAP for a given topic. We do this in order to investigate the question of absolute performance on the CSP algorithm.

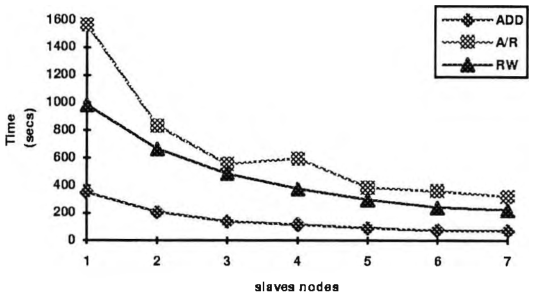


Fig 10-22. ZIFF-DAVIS [*Replication*]: restricted iterations on CSP algorithm, average elapsed time in seconds (AP3000)

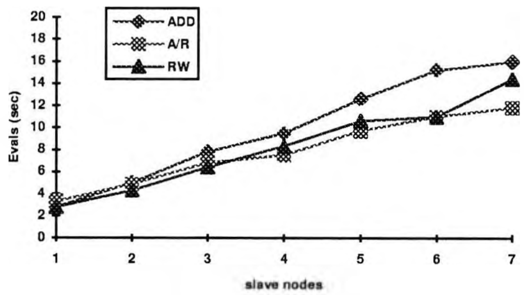


Fig 10-23. ZIFF-DAVIS [*Replication*]: restricted iterations on CSP algorithm, throughput in evals per second (AP3000)

Restricting the iterations on the CSP algorithm has a dramatic effect on time (see fig 10-22): run times on parallel runs are much reduced from those declared in fig 10-1 to 10-3. Unlike the unrestricted runs, there is a clear linear reduction in run times, resulting in a linear increase in throughput for all operations. Throughput on restricted runs is generally better than for unrestricted runs, particularly on more slave nodes (see fig 10-23). The number of evaluations for the *add remove* operation varies, but the other two term operations show linear increase in the number of evaluations with more slave nodes: for example the workload increases from 971 evaluations on 1 slave node to 1141 evaluations on 7 slave nodes on the *add only* operation.

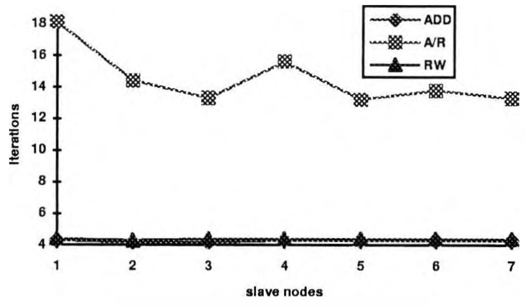


Fig 10-24. ZIFF-DAVIS [*Replication*]: restricted iterations on CSP algorithm, average outer iterations (AP3000)

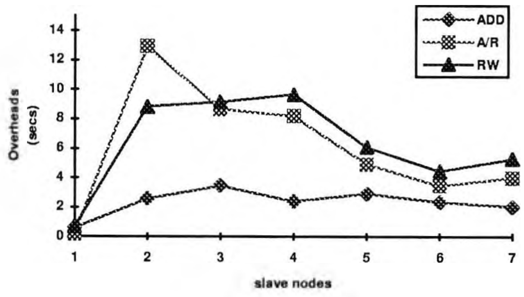


Fig 10-25. ZIFF-DAVIS [*Replication*]: restricted iterations on CSP algorithm, overheads in seconds (AP3000)

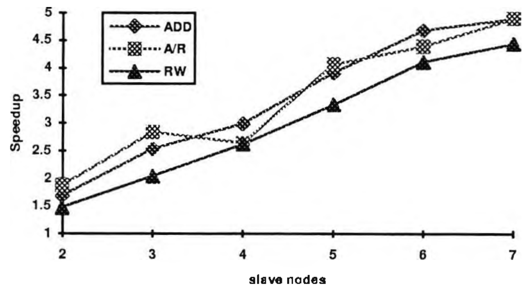


Fig 10-26. ZIFF-DAVIS [*Replication*]: restricted iterations on CSP algorithm, speedup (AP3000)

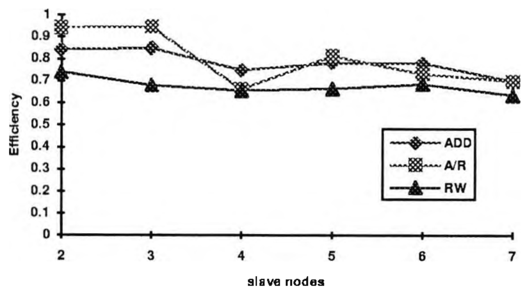


Fig 10-27. ZIFF-DAVIS [*Replication*]: restricted iterations on CSP algorithm, parallel efficiency (AP3000)

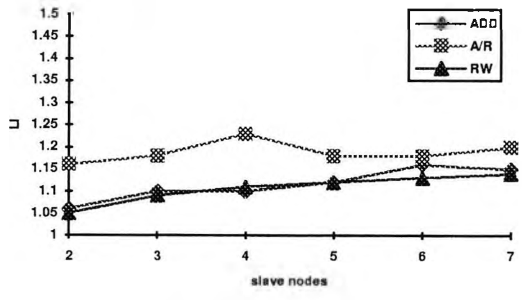


Fig 10-28. ZIFF-DAVIS [*Replication*]: restricted iterations on CSP algorithm, load imbalance (AP3000)

Restricting the number of iterations on CSP does not necessarily mean that the number of iterations taken will the identical to that of CAP (see fig 10-24). This is because iterations on an individual topic may take fewer outer iterations to complete term selection on CSP. In practice this effect does not make much difference to runs on *add only* and *add reweight* operations, but runs on *add remove* are affected and the number of iterations vary erratically. As with unrestricted iteration runs overheads per iteration are high, but are more significant in restricted iteration runs (see fig 10-25).

Fig 10-26 demonstrates that relative speedup is available on CSP with any operation, but there is still a restriction on gains in using parallelism. This is because the extra workload which CSP with restricted iterations imposes on CAP is an important factor in the loss in parallel performance: particularly for *add only* and *add reweight* operations. Parallel efficiency levels are much higher in restricted iterations than in unrestricted runs (see fig 10-27): all runs record efficiency above 0.6, while efficiency in unrestricted runs can be well below this figure. Load imbalance has the same effect on restricted iteration runs as it has on the unrestricted runs (see fig 10-28): that is imbalance is a minor problem but could become a significant one using a parallel machine with more slave nodes.

10.3.1.7 Summary of *Replication* Retrieval Efficiency Results

There are three factors which can affect the performance of the term selection algorithms using parallelism. The first of these is extra workload: more evaluations are performed in most cases for both CFP and CSP which imposes a computational load over and above that of the sequential (1 slave node) runs. The overhead at the synchronisation point required to resolve the difference between data on slave nodes is a significant bottleneck, particularly for the CFP algorithm. An increase in load imbalance is recorded for all runs with larger parallel machine sizes: less important than the other two but a potential source for improvement. These three factors affect the different algorithm in different ways.

Using parallelism with the Find Best algorithm works well with any of the studied term operation techniques. Linear time reduction and a near linear level of speedup and efficiency are recorded with Find Best, as well as an increased level of evaluation throughput with increasing numbers of slave nodes. Overheads are constant, but significant with increasing numbers of slave nodes.

Given the evidence above it is hard to justify the form of parallelism specified in *replication* as applied to the CFP algorithm. While there is time reduction with any of the chosen operations, much of the selection efficiency advantage gained can be explained by the reduction in average iterations per topic. However some of the selection efficiency loss can be explained by an increase in the workload for increasing numbers of slave nodes: there is clearly an offset between number of iterations and evaluations required per topic. While load imbalance appears to be a problem, the overheads on the CFP algorithm are a significant bottleneck due to the number of iterations needed to optimise a query. Comparing the results on an operation by operation basis, CFP does not match the selection efficiency gained either in terms of time or evaluation throughput with the other term selection methods.

The results for *Replication* using CSP with *add only* and *add reweight* demonstrate that more of the search space is examined than CAP with those same operations, while *add/remove* is more non-deterministic and hence average elapsed times are erratic. Restricting the iterations in order to examine the relative level of selection efficiency demonstrates that some speed improvement is found with CSP: despite examining a little more of the search space. It also possible to obtain slight selection efficiency improvements even when examining far more of the search space. Overheads tend to decline in importance with CSP, but are larger in real terms with this method because of the amount of data which needs to be exchanged between the master and slave nodes at the synchronisation point.

10.3.2 *Retrieval Efficiency Results: On-the-fly Distribution*

We consider the results for *on-the-fly* distribution. Recall that in *on-the-fly* the master node distributes data from a centrally held inverted file to slave nodes (see chapter 4, sub-section 4.8.3.5 and fig 4-17).

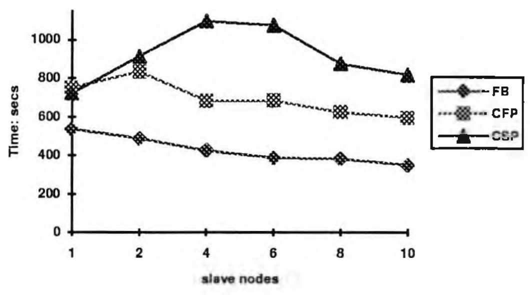


Fig 10-29. ZIFF-DAVIS [*On-the-fly*]: *add only* average term selection elapsed time in seconds (Network)

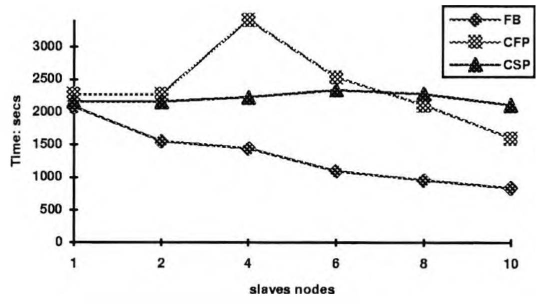


Fig 10-31. ZIFF-DAVIS [*On-the-fly*]: *add reweight* average term selection elapsed time in seconds (Network)

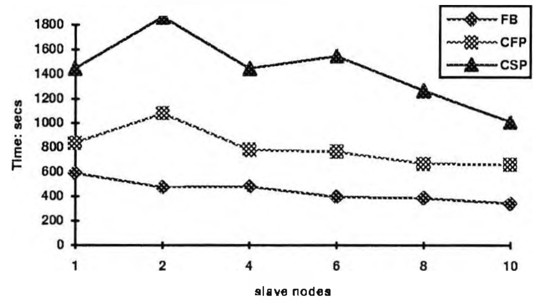


Fig 10-30. ZIFF-DAVIS [*On-the-fly*]: *add remove* average term selection elapsed time in seconds (Network)

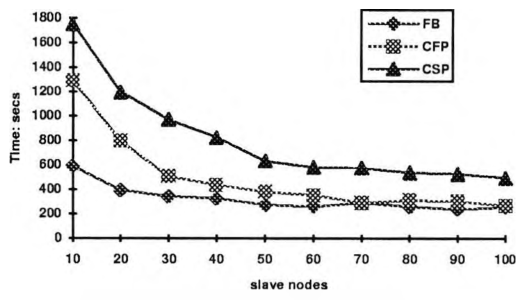


Fig 10-32. ZIFF-DAVIS [*On-the-fly*]: *add only* average term selection elapsed time in seconds (AP1000)

The results on term selection are disappointing with little if any speed advantage shown by the deployment of parallelism (see figs 10-29 to 10-32). Given the level of parallelism used the results are particularly bad for runs on the AP1000: we only did runs on *add only* as a result (see fig 10-32). The effect on parallel measures is significant with quite a few runs registering slowdown from 1 slave node runs (see appendix A3, fig A3-1). The main reason for these poor results can be seen in figs 10-33 to 10-36 below.

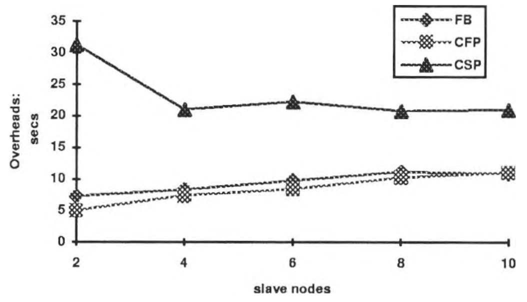


Fig 10-33. ZIFF-DAVIS [*On-the-fly*]: *add only* average overheads per outer iteration for term selection (Network)

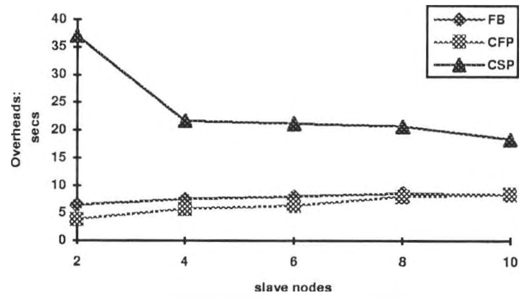


Fig 10-35. ZIFF-DAVIS [*On-the-fly*]: *add reweight* average overheads per outer iteration for term selection (Network)

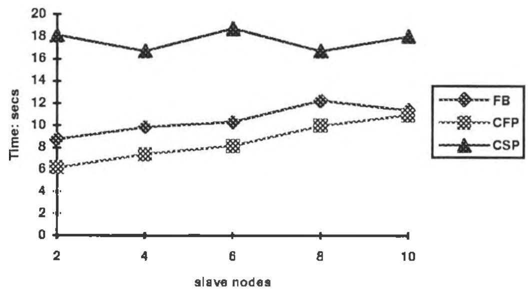


Fig 10-34. ZIFF-DAVIS [*On-the-fly*]: *add remove* average overheads per outer iteration for term selection (Network)

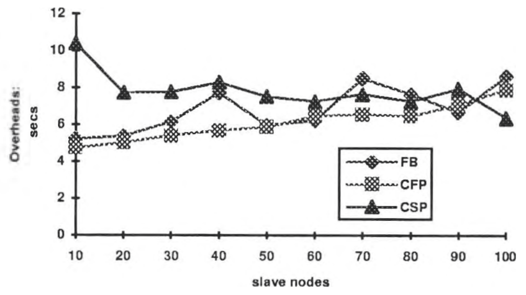


Fig 10-36. ZIFF-DAVIS [*On-the-fly*]: *add only* average overheads per outer iteration for term selection (AP1000)

The overheads on *on-the-fly* are far larger than those of *replication* and form a significant part of overall time. The trend on the Network runs is upwards in most cases for both FB and CFP algorithms, whereas overheads for CSP are fairly constant. The higher network capacity on the AP1000 is able to better deal with the transfer of data, but overheads are still so high that when using larger number of slave nodes more time is spent at the synchronisation point than doing useful work. As shown in Rungsawang et al (1999) network bottlenecks are a serious problem. The average number of iterations needed per topic can be seen in the appendix A3, fig A3-4. Tables 10-1 and 10-2 shows a further bottleneck with *on-the-fly* distribution.

slave nodes	FIND BEST (FB)			CHOOSE FIRST POSITIVE (CFP)			CHOOSE SOME POSITIVE (CSP)		
	ADD	A/R	RW	ADD	A/R	RW	ADD	A/R	RW
1	21.7	32.2	29.8	27.0	31.5	20.84	23.0	35.3	19.9
2	19.3	35.7	30.0	25.6	31.8	19.47	19.5	41.6	22.3
4	20.0	36.2	31.0	31.2	31.4	19.81	19.2	31.9	20.4
6	21.5	37.7	23.6	34.6	30.3	19.47	25.0	39.4	23.8
8	19.4	37.5	19.6	34.7	39.6	21.63	21.5	30.1	24.1
10	20.5	32.5	18.5	37.8	44.6	27.42	28.7	32.4	21.4

Table 10-1. ZIFF-DAVIS [*on-the-fly*]: Load data overheads in seconds (Network)

Slave nodes	FB	CFP	CSP
10	23.2	22.6	28.1
20	21.7	22.4	23.7
30	23.7	23.1	22.0
40	28.7	24.4	29.4
50	21.4	23.0	23.5
60	22.6	24.9	22.6
70	29.6	22.2	25.3
80	25.9	24.2	22.7
90	23.4	24.8	28.4
100	33.4	29.8	22.0

Table 10-2. ZIFF-DAVIS [*on-the-fly*]: Load data overheads in seconds for *add only* (AP1000)

When using a strategy which holds one copy of the inverted file, the posting lists for all the terms in the term pool must be built in the *master* node for distribution to the slave nodes. The data shown in tables 10-1 and 10-2 is a sequential part of the algorithm to which parallelism cannot be applied. While the load times are not particularly heavy, being around 20 to 30 seconds for most runs, they are significant for runs which utilise many tens of slave nodes. Distributing this work in *replication* holds advantages for parallelising data loads.

Given the increase in processor speeds in the recent past, it is difficult to justify using parallelism on a distribution scheme such as *on-the-fly* which requires large data transfers at the synchronisation point. The *replication* strategy discussed above which keeps data local is able to reduce the amount of data transferred considerably and is able to parallelize posting list builds for terms being processed.

10.3.3 *Retrieval Effectiveness Results: On-the-fly Distribution*

In this section we examine the retrieval effectiveness result for routing using the metrics declared in chapter 3, sub-sections 3.4.2.1 and 3.4.2.2, using the evaluation functions for filtering described in sub-section 3.4.2.3. Note that the best results given in the tables are highlighted in bold.

10.3.3.1 Find Best

The routing results given in table 10-3 are very good: selection precision/recall are very high indeed and the test set results are on or around the maximum you would expect with TREC data. The extra iterations in the parallel versions produced precision oriented selection results, but the sequential versions yield better results on the test set. However the difference is not significant. On a term operation basis *add only* and *add/remove* both produce better test set results than *add reweight*: there is a slight overfit problem with the latter. With respect to the filtering results it was found that *add/remove* provided the highest utility for all the functions. There was slight variation in both average precision and utility function results when applying the *add reweight* operation: all variations are significantly less than 1%.

MEASURES	ADD ONLY		ADD/REMOVE		ADD REWEIGHT*	
	SEQ	PAR	SEQ	PAR	SEQ	PAR
Select Recall	0.924	0.922	0.923	0.920	**0.929**	**0.929**
Prec.	0.723	0.727	0.723	0.727	0.772	0.772-**0.778**
Test set Recall	**0.837**	0.834	0.836	0.833	0.830	0.830
Prec.	**0.462**	0.454	**0.462**	0.454	0.433	0.425-0.427
U1	11.95	11.74	**12.11**	11.89	11.26	11.05 - 11.11
U2	18.95	18.74	**19.16**	19.0	17.37	17.16 - 17.21
U3	311.6	311.6	**314.9**	**314.9**	276.1	276.2 - 276.3

Table 10-3. ZIFF-DAVIS: Find Best routing and filtering results
(* results gathered on network version only)

10.3.3.2 Choose First Positive

With *add only* operation on routing, no pattern in either increase or decrease was found and recall/precision did not vary much: either on the selection or test set database (see table 10-4). Evaluation on the Selection databases showed better average recall/precision over multiple slave nodes than one slave node, but the difference is not significant. The best figures for precision on the selection database were 0.741 for 6 slave nodes, while for the test set database it was 0.483 on 2 slave nodes. The filtering results do not show any pattern of increase or decrease, the best utility results being recorded on multiple slave node runs.

slave nodes	Select recall	Prec.	Test set recall	Prec.	U1	U2	U3
1	0.906	0.714	0.859	0.461	12.79	18.63	275.0
2	0.912	0.732	0.860	**0.483**	**13.11**	19.42	267.6
4	0.916	0.738	0.855	0.459	12.21	18.37	294.8
6	0.917	**0.741**	0.836	0.447	12.21	18.21	**313.7**
8	0.917	0.735	**0.877**	0.472	11.0	17.89	303.7
10	0.915	0.733	0.862	0.480	13.05	**19.84**	303.7
20	0.916	0.740	0.871	0.474	12.16	18.89	310.1
30	0.915	0.731	0.809	0.447	11.37	17.58	299.4
40	**0.925**	0.732	0.843	0.475	12.21	19.21	301.3
50	0.917	0.730	0.831	0.460	12.11	18.89	230.3
60	0.922	0.734	0.807	0.439	11.47	18.32	259.5
70	0.923	0.723	0.831	0.458	12.53	19.53	313.0
80	0.924	0.732	0.832	0.471	12.47	19.74	302.9
90	0.923	0.726	0.831	0.462	12.42	18.79	312.7
100	0.922	0.728	0.834	0.452	12.11	18.58	308.4

Table 10-4. ZIFF-DAVIS: Routing/filtering effectiveness results for CFP *add only* operation

slave nodes	Select recall	Prec.	Test set recall	Prec.	U1	U2	U3
1	**0.923**	**0.754**	0.872	**0.487**	13.11	20.32	293.2
2	0.914	0.746	0.864	0.465	12.58	19.42	292.9
4	0.919	0.753	**0.884**	0.486	13.11	**21.37**	302.2
6	0.920	0.752	0.833	0.455	12.47	18.47	314.2
8	0.916	0.744	0.874	0.482	11.89	19.37	253.4
10	0.911	0.739	0.858	0.480	**13.16**	19.74	**317.5**

Table 10-5. ZIFF-DAVIS: Routing/filtering effectiveness results for CFP *add/remove* operation

slave nodes	Selection Recall	Prec.	Test set Recall	Prec.	U1	U2	U3
1	0.913	0.717	0.843	0.422	8.58	14.42	208.3
2	0.912	0.745	0.858	0.444	10.79	17.11	255.2
4	0.922	0.754	0.859	0.451	10.26	16.63	238.2
6	0.915	0.762	**0.871**	**0.458**	11.11	17.11	258.1
8	0.919	0.767	0.847	0.440	10.47	17.00	248.3
10	**0.934**	**0.777**	0.858	0.451	**12.42**	**18.53**	**298.4**

Table 10-6. ZIFF-DAVIS: Routing/filtering effectiveness results for CFP *add reweight* operation

Using parallelism with *add/remove* operation appears to be detrimental to the CFP algorithm with respect to routing (see table 10-5): only test set recall yields a better result using multiprocessors than the uniprocessor run. The variations between runs are very small and not statistically significant. The filtering results by contrast do show that better results can be obtained using multiprocessors: the highest utilities for U1 and U3 where found on the 10 slave node run.

Unlike the other two term operations CFP with *add reweight* does show a slight linear increase in selection precision with increasing numbers of slave nodes (see table 10-6): this improvement is not wholly reflected in test set results. The best selection results for both recall and precision are found using 10 slave nodes, which also yield the highest figures for all utility functions. All routing and filtering results on the parallel runs yield better figures than the uniprocessor run. From the figures above we can state that finding better terms earlier with the parallel CFP method using any operation does not yield a significant improvement or drop in retrieval effectiveness for any evaluation method studied.

10.3.3.3 Choose Some Positive

slave	**Selection**		**Test set**		**U1**	**U2**	**U3**
nodes	**Recall**	**Prec.**	**Recall**	**Prec.**			
1	0.913	0.742	0.863	0.473	12.37	17.63	262.1
2	0.913	0.743	0.857	0.455	12.47	17.11	268.6
4	0.916	0.749	0.856	0.450	11.68	18.11	290.6
6	0.916	0.736	0.849	0.478	14.0	**21.16**	305.4
8	0.921	0.750	0.862	0.450	12.0	17.68	287.0
10	**0.923**	0.756	0.852	**0.481**	12.74	18.79	294.3
20	0.916	0.754	0.858	0.465	11.95	18.84	114.1
30	0.919	**0.760**	0.828	0.439	12.58	17.84	262.6
40	0.918	0.747	0.837	0.458	12.58	18.95	279.1
50	0.905	0.747	0.829	0.445	11.95	17.74	289.0
60	0.913	0.746	0.829	0.447	12.95	18.53	293.4
70	0.907	0.744	0.856	0.445	11.74	18.47	288.5
80	0.905	0.737	**0.873**	0.455	12.68	18.53	**312.5**
90	0.915	0.752	0.852	0.448	**13.79**	18.05	284.1
100	0.918	0.754	0.843	0.459	12.37	18.42	280.3

Table 10-7. ZIFF-DAVIS: Routing/filtering effectiveness results for CSP *add only* operation

It was found that differing the number of slave nodes in CSP from CAP (i.e. 1 slave CSP = CAP) using *add only* operation had an effect on both the recall and precision averages on selection terms over all topics: at least one parallel run beat the sequential run on all evaluation measures. No pattern was found in either average recall or precision (see table 10-

7) for either selection or on the test set. There is no correlation between selection and test set with overall recall and precision. Two large slave node set runs produced the highest utility for U1 and U3 filtering functions. It should be noted that any improvement found was not significant.

slave nodes	Selection Recall	Prec.	Test set Recall	Prec.	U1	U2	U3
1	0.920	**0.782**	**0.866**	0.479	12.79	18.37	295.5
2	0.919	0.774	0.864	0.452	10.95	18.05	263.2
4	0.918	0.778	0.849	0.459	**14.0**	18.84	**305.2**
6	**0.925**	0.760	0.825	0.452	12.32	19.26	302.0
8	**0.925**	0.770	0.860	0.452	10.84	17.95	296.2
10	0.922	0.772	**0.866**	**0.487**	12.42	**19.37**	277.8

Table 10-8. ZIFF-DAVIS: Routing/filtering effectiveness results for CSP *add/remove* operation

Table 10-8 shows that no pattern of improvement or degradation of retrieval effectiveness was found in *add/remove* CSP runs compared with the CAP run. The best selection precision was found in the CAP run, but the parallel run with 10 slave nodes did yield slightly better average precision than any other run. At least one parallel run produced higher average filtering utility results than the uniprocessor run.

slave nodes	Select recall	Prec.	Test set recall	Prec.	U1	U2	U3
1	0.906	0.735	0.839	**0.432**	8.84	15.58	217.7
2	0.912	0.757	**0.841**	0.431	**11.32**	**16.42**	224.1
4	0.905	0.760	**0.841**	0.411	9.16	14.63	207.1
6	0.913	0.755	0.825	0.425	9.47	15.53	222.2
8	0.914	0.766	0.831	0.408	10.0	14.89	229.2
10	**0.917**	**0.769**	0.829	0.409	9.84	15.37	**232.0**

Table 10-9. ZIFF-DAVIS: Routing/filtering effectiveness results for CSP *add reweight* operation

Table 10-9 shows the *add reweight* operation results yield results which are opposite to *add remove*: the best selection precision is at 10 slave nodes, while the best test set precision is on 1 slave node. All selection precision results are better on the parallel runs than the uniprocessor run. There does appear to be an overfitting problem with this method. Filtering results show that higher utility values are gained on parallel runs, but U1 and U2 only yield the best on two slave nodes. To sum up the CSP results, one might expect the increase in the number of iterations to give improved effectiveness. These results suggest parallelism can cause a small improvement but the slave node set size to use for any parallel run is difficult to determine: any retrieval effectiveness difference found by using parallelism is minimal.

10.3.4 *Ziff-Davis Experiment Summary*

With respect to time and studied performance improvement figures for parallelism clearly *replication* is by far a better method for improving term selection times for the query optimisation methods studied in this research. There are two main reasons for the advantage found in *replication*: overheads per iteration and load times. The most important of these is overheads per iteration, that is the sequential bottleneck for the term selection algorithm under discussion. While *replication* does not completely eliminate the overheads, it does substantially reduce them because the amount of data that needs to be transferred between the master and slave nodes is much reduced. With *replication* only the query is broadcast and any the data for subsequent terms chosen by a term selection algorithm exchanged by *master/slave* nodes is restricted to the identifier(s) of that term. Another minor advantage is that the load time is parallelised in *replication* and is not a bottleneck. Given that the parallelism in *on-the-fly* yields very little if any time advantage (and in fact may lead to slowdown) over uniprocessor we assert that it is not a viable option. *Replication* however does show promise for two of the algorithms, FB and CSP and our results show that this form of parallelism does lead to selection efficiency improvement. The best performing algorithm in speed terms comparing on an operation by operation basis was Find Best when using *replication* parallelism.

Examining the retrieval effectiveness results we conclude that although parallelism may bring benefits to the term selection algorithms CFP and CSP, there is no pattern and no guarantee that parallelism brings benefits with increasing numbers of slave nodes from the evidence of our results. Results on all runs were on the high side. There are cases where parallel runs yield results that are not as good as the corresponding sequential run (for a given algorithm and operation pair). However any increase or decrease in retrieval effectiveness is not significant, and these experiments do not demonstrate that examining a larger part of the search space will bring benefits to in terms of retrieval effectiveness for any of the term selection algorithms discussed in this chapter. The best performing algorithm tended to be CFP, while *add/remove* operation seemed to produce the best results in conjunction with the CFP algorithm. The best performing parallel slave node set size with respect to effectiveness tended to be 4 and 10.

10.4 TREC-8 EXPERIMENTS

TRACK RUN-ID	SUB-TRACK	ALGORITHM	OPERATION
plt8f1	BATCH FILT.	FIND BEST	ADD/REMOVE
plt8f2	BATCH FILT.	FIND BEST	ADD/REWEIGHT
plt8r1	ROUTING	FIND BEST	ADD/REMOVE
plt8r2	ROUTING	FIND BEST	ADD/REWEIGHT

Table 10-10. Details of TREC8 filtering track runs

Given that the main thrust of our research is to improve the elapsed times of the algorithms for tasks we decided to try Find Best using *add remove* and *add reweight* operations on our TREC8 experiments (MacFarlane et al, 2000a). Both operations are more computationally intensive than *add only* and can yield better retrieval effectiveness. This allowed us to demonstrate that the data distribution strategy found to be useful on Ziff-Davis data, namely *replication*, could improve query optimisation times on a much larger training set using a larger parallel machine. We entered four runs for TREC 8: details of these can be found in table 10-10.

We treated the databases differently in the different sub-tracks. With batch filtering runs we did extraction of terms and term selection on one database: this was because of the small number of relevant documents available in the training set. However with Routing we were able to do extraction of terms on one database and term selection on another as per Okapi experiments (Beaulieu et al, 1997). The number of relevant documents in the routing training set allowed us this flexibility (the main reason for splitting the training set when using the term selection algorithms is to reduce the overall level of overfitting). Topics without relevant documents were not treated differently from topics with relevant documents as we were testing the parallelization of the query optimization algorithms.

All term selection runs were optimized using TREC average Precision: we tried using the utility function LF1 but the results were poor. This confirms the Okapi experimental result which suggest that average Precision is a good predictor for other measures, but the other measures do not predict each other well. All Batch Filtering runs were optimized for the U1 utility function.

We did some initial experiments with TREC-7 AP Filtering track data as a "dry run": the results are reported briefly in each section below. We split the discussion of TREC-8 experiments into two sections: one of effectiveness and one of efficiency. Each section has discussion on the sub-tracks entered: batch filtering and routing.

10.4.1 *Retrieval Efficiency Results*

10.4.1.1 Batch Filtering

The average query selection time for the AP data set was 115 seconds, taking on average 26.56 iterations to select an average of 28.2 terms. The results submitted on the FT data set for TREC-8 are in stark contrast. For run plt8f1 the average term selection time per topic was 6.7 seconds with an average of 8 iterations choosing an average of 9.3 terms. Run plt8f2 was slightly more costly computationally taking 19 seconds per topic on 8.5 iterations with an average of 10 terms chosen per topic. The LI was very poor for FT data: a LI of 1.65 was recorded for plt8f1 while for plt8f2 the figure was 2.15. The LI for AP data was 1.46, an improvement on the FT data figures but still not particularly good. The reason for the reduced load balance in these experiments is that some nodes had terms which were far more costly to evaluate than others: even though the slave nodes were given virtually the same number of terms to inspect. In the context of time it would not therefore seem to be any use in applying parallelism to the Okapi term selection algorithms for smaller databases where there are only a limited set of relevance judgments. It is important to try and find the accumulation level for relevant documents on topics where term selection could be applied and parallelism could be considered. We did not do any runs on lesser numbers of slave nodes as a consequence.

10.4.1.2 Routing

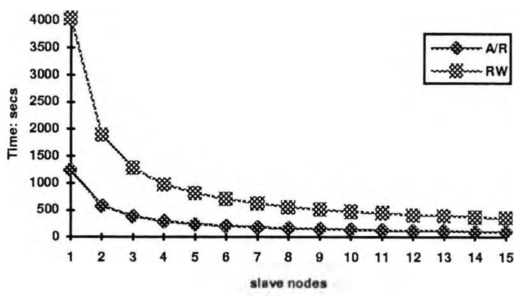


Fig 10-37. TREC8 Find Best experiments: average elapsed time in seconds for term selection

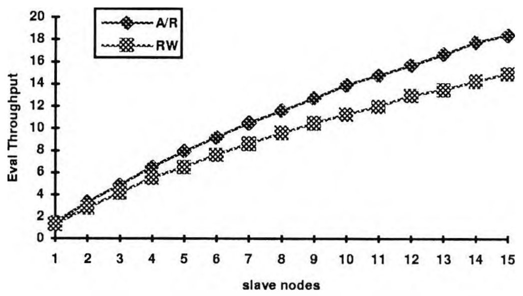


Fig 10-38. TREC8 Find Best experiments: evals per second throughput for term selection

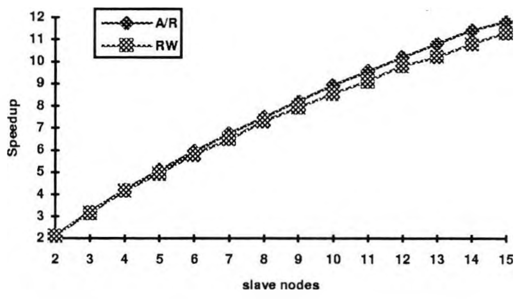


Fig 10-39. TREC8 Find Best experiments: speedup for term selection

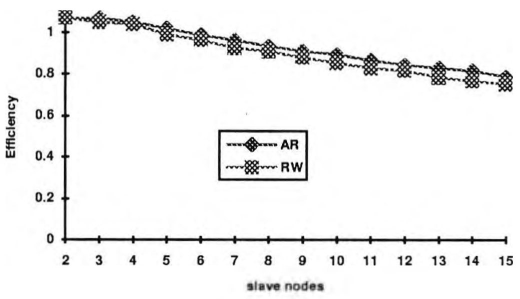


Fig 10-40. TREC8 Find Best experiments: parallel efficiency for term selection

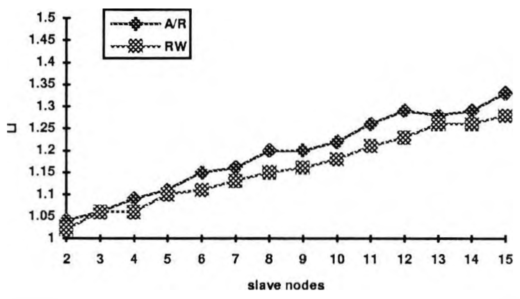


Fig 10-41. TREC8 Find Best experiments: load imbalance for term selection

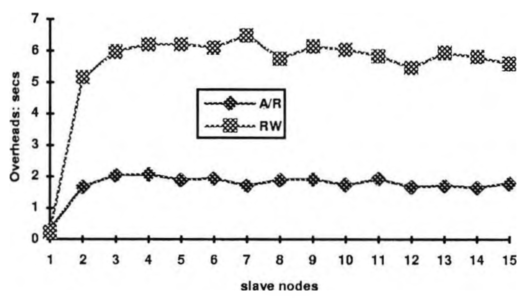


Fig 10-42. TREC8 Find Best experiments: iteration overheads for term selection

On 15 slave nodes the average term selection time for AP data was 23 minutes with an average of 49.5 iterations choosing 51.5 terms on average. The average term selection times for FT data were much smaller: taking 1.75 minutes for run plt8r1 and 6 minutes for plt8r2. The number of terms chosen was an average of 21 for plt8r1 and 27 for plt8r2. The number of iterations on FT data was also much reduced being on average 20 for plt8r1 and 25 for plt8r2. The LI for add with re-weight operation was much better than for the *add remove* operation: term selection on AP data yielded a LI on 1.25 while for plt8r2 the figure was 1.28. Run plt8r1 yielded a LI of 1.33 by contrast. The results with respect to efficiency are far superior in routing than batch filtering: this is largely due to the size of the data set used and the number of relevance judgements available in the routing task compared with filtering. The size of the data set used has a considerable impact on load imbalance. The size of the collection would be therefore a factor when examining the viability of deploying term selection algorithms. Results from runs which use lesser numbers are shown in figs 10-37 to 10-42.

The selection efficiency using Find Best with *add/remove* operation is very encouraging. Not only do we obtain linear time reduction with the method, we also get a good level of speedup and efficiency. The throughput is impressive and demonstrates a linear increase with more nodes. Overheads are constant and increase in significance with more slave nodes, and load imbalance is clearly a problem with increased levels of parallelism. The average number of iterations per topic is 20.3.

Find Best with *add reweight* is far more costly than add only: the total time to optimise queries on all 50 topics using 1 slave node was 56.1 hours, just over three times the time needed for *add only* operation. However there is clearly a selection efficiency improvement with increasing numbers of slave nodes, although evaluation throughput is not as good as *add only* and overheads are higher (if constant). Load imbalance is also a problem, although not so pronounced.

The results show that the method of parallelism found useful in Ziff-Davis experiments was also good at speeding up optimisation times for queries on the much larger TREC8 training set. There is clearly a need to tackle the problem with load imbalance found in the results given in this section (see the thesis conclusion on a discussion of this subject).

10.4.2 *Retrieval Effectiveness Results*

10.4.2.1 Batch Filtering

TRACK RUN-ID	**SELECTION (Recall/Prec)**	**TEST DB (Recall/Prec)**	**AVERAGE EVALUATIONS PER TOPIC**	**AVG SCALED LF1**
plt8f1	0.856/0.843	0.142/0.280	907	0.354
plt8f2	0.855/0.849	0.149/0.287	2022	0.376
AP Run	0.852/0.816	0.250/0.118	5764	-

Table 10-11. Details of TREC8 batch filtering efficiency results

Both submitted filtering runs were optimised for the LF1 utility function. We present the results in table 10-11: the average scaled utility function used is from Hull (1999) and declared in chapter 3, sub-section 3.4.2.3. The Recall/Precision for the selection runs are all very good indeed: the number of relevant documents per topic is 51 compared with just under 11 per topic for the FT data therefore our runs for TREC-8 have done better with less data. This is also true of Precision on the FT test database but not true of Recall. Our filtering runs for TREC-8 sacrifice recall for precision. The precision for filtering is comparable with routing results (see table 10-12 below) and much higher than was expected given the type of method used for filtering and the number of relevant documents available. Comparing *add/remove* operation to *add reweight* we found an increase of 2.5% for the former over the latter: this increase is not particularly significant given that *add reweight* need 2.2 times the average evaluations per topic than *add/remove*. The increase in scaled average utility was more significant: using *add reweight* operation yielded a 6% advantage over Add/Remove.

10.4.2.2 Routing

TRACK RUN-ID	SELECTION (Recall/Prec)	TEST DB (Recall/Prec)	AVG EVALS PER TOPIC
plt8r1	0.873/0.696	0.858/0.286	1932
plt8r2	0.887/0.734	0.845/0.288	5364
AP Run	0.824/0.608	0.543/0.286	5481

Table 10-12. Details of TREC8 routing efficiency results

As with Batch Filtering we did one test run on AP data with Find Best using the Add reweight operation. The precision/recall for term selection was high with values of 0.824 and 0.608 respectively. The results on the test collection compared favourably with participants of the TREC-7 routing sub-track: our average precision of 0.286 was better than 5 of the 10 runs submitted by participants of that track. Recall/precision for the TREC8 runs on selection data was very good indeed with recall just under 0.9 and precision around 0.7. Results on the test database are very good on recall which is about 0.85, while precision was adequate at around 0.28 (see table 10-12). Comparing *add reweight* operation as against *add remove* we found that *add reweight* did bring benefits over *add remove* but the gain was only 0.7%. This figure is not much of an increase for the extra work needed in *add reweight* where a factor of 2.78 more evaluations where needed over *add remove*. With respect to precision on run plt8r1, 20/50 topics were better than median while two equalled the best: topics 355 and 380. For run plt8r2 15/50 were better than median and the same two topics as in plt8r1 equalled the best. It should be noted however that these two best yielding precision topics only contained one relevant document each. In two of the topics 387 and 394 run plt8r2 recorded the best average precision. Overall the results were acceptable, if a little disappointing compared with other participants in the TREC-8 routing sub-track.

10.5 CONCLUSION

We have found a method of parallelism together with a data distribution method (*replication* of inverted file data) which allows us to both speed up and examine more of the search space for the heuristics examined. This was done by focusing on the main task, namely the evaluation of terms during term selection. In conjunction with replication we have shown with Ziff-Davis and TREC8 data that the speed advantage found with the Find Best selection method is significant. We have demonstrated that we are able to examine more of the search space with the CSP algorithm and still reduce the overall time for query optimisation. We believe it is possible to improve the selection efficiency of both methods using some form of dynamic re-distribution technique for terms in the query. Experiments with CFP are less conclusive and show difficulties with load balance, overheads, time and evaluation throughput. It may be possible to improve the load balance of CFP but only at a larger overhead cost. We have demonstrated that keeping the inverted file at the master and broadcasting information as and when it is needed is not a viable option for applying parallelism to the optimisation of routing/filtering queries. The synthetic model was successfully able predict that *replication* is a superior method to *On-the-fly* distribution, but not that the latter would perform so poorly. As found in our probabilistic search experiments, modelling communication is hard to do. We can make a further statement on the synthetic models, given the evidence found with *On-the-fly* distribution. The *TermId* partitioning method requires more communication at the synchronisation point than *On-the-fly* distribution and is therefore not a viable method.

Whilst we have shown using Ziff-Davis data that examining the search space can improve effectiveness, it is also clear that doing so can actually harm effectiveness as well. However any gain or losses are statistically insignificant and we have only demonstrated that examining more of the search space can find different maxima in that space. It may be possible to use more powerful machine learning or pattern recognition techniques to find the best maxima out of the many that are available (see the conclusion for a discussion on this).

Chapter 11

Summary and Conclusions

11.1 OVERVIEW OF DATA DISTRIBUTION METHODS

Recall that the primary aim of this thesis is to examine the performance of various IR tasks using various data distribution methods on parallel computers to determine which of the distribution methods is best. Our criterion is that a given distribution method should demonstrate either a decrease in elapsed time (speedup) or the ability to search a larger database (scalability and scaleup). Overall the best performing data distribution method was *DocId* partitioning, for all the tasks discussed in this thesis apart from routing/filtering: this is the main contribution of this thesis. This conclusion was reached by a combination of argument, theoretical modelling and empirical results. Some of the distribution methods such as on-the-fly and replication were not relevant to many tasks for different reasons. For example on-the-fly distribution is irrelevant method for the indexing task (our consideration of distribution is on inverted file data not raw text). We chose not to implement partitioning methods for the routing/filtering task as we believe that load balancing would be a serious problem in *TermId*, while *DocId* would only be able to speed up a single evaluation. The synthetic model we have derived showed that in theory, *DocId* partitioning was better for most tasks, and this was confirmed by empirical results.

What are the reasons for the success of *DocId* partitioning for most tasks and replication for routing/filtering? It would help if we considered what each task does, i.e. what is the computation of the task. For most tasks the emphasis is on document computation, e.g. index then search for a document. For routing/filtering the emphasis is on term computation, e.g. how well does a given term do when we evaluate it against the training set? We wish to keep data local to a node and reduce the amount of data which must be moved: the concept of locality of reference is important here. For most tasks keeping document data in one location is important, while for routing/ filtering we need access to term data on one node. Any attempt to use an inappropriate data distribution method for a given task will increase the level of communication for that task to such an extent that there is little or no advantage to be gained from using parallelism.

11.2 DISCUSSION OF INDIVIDUAL TASKS

As well as the overall contribution to the field of parallelism and IR discussed above we have also made some individual contributions to tasks studied in this thesis. Some of these relate to the secondary aims of our thesis These are as follows:

- A high bandwidth network is essential if *TermId* partitioning indexing is to be efficient. To the best of our knowledge we are the first to make a direct comparison between partitioning methods in the indexing task.
- We have found in the search task that query size does have an effect on performance using the partitioning methods studied, but technological factors such as sorting the data to generate the final ranked result are more important.
- We have been unable to demonstrate that examining more of the search space in the passage retrieval task brings any benefits, but we have shown that the BM25 weighting function does its job well and in conjunction with passage retrieval on a limited set of documents, can increase retrieval effectiveness. To the best of our knowledge we are the first to make a direct comparison between partitioning methods in this task or make a practical attempt at using parallelism to improve retrieval efficiency of the task.
- To the best of our knowledge we are the first to make a direct comparison between partitioning methods in the index update task. The *DocId* partitioning method reduces the amount of index data per node with increasing parallel machine size: this is due to the qualities of keyword blocks where hit terms will be interspersed amongst less frequent terms. We have identified a number of potential logical errors during transaction processing on inverted files in the presence of an incorrect or non-existent concurrency control mechanism which may affect retrieval effectiveness.
- In the routing/filtering task we have demonstrated that more of the search space can be examined, but have found this does not always increase retrieval effectiveness. We have found that using parallelism in the way we describe, finds different maxima which yield approximately the same level of retrieval effectiveness. To the best of our knowledge we are the first to make a direct comparison between data distribution methods in this task or make an attempt at using parallelism to improve efficiency of term selection for the task. We have proposed a method of term selection, namely Choose Some Positive (CSP), which has not been used in previous Okapi experiments.

• We have derived a synthetic model for performance for more than one task which is able to distinguish between the distribution methods discussed in this thesis (see section 11.3 for a fuller discussion on the models).

How generic are these conclusions? What would happen to the comparative performance on the data distribution methods if we used compression techniques for inverted lists or query processing optimisation techniques for example? In some cases it may be difficult to make any sensible statement without forming some theoretical model for comparison and testing such with empirical results. We can however state the following:

• With the indexing and index update tasks our conclusions are very generic. No matter what type of index used, indexing *TermId* would always need to communicate more than *DocId* indexing and apply more computational effort to produce the final index. Using *DocId* in index update we would expect better performance than *TermId* as much less data needs to be moved, providing our keyword block mechanism is utilised.

• Our conclusions with probabilistic search apply only to term weighting models of any type (such as the vector space model). They do not apply to Boolean or proximity models which do not require sorts to organise the final results (though one would still need to pass up large sets for central merging in *TermId* partitioning with those models).

• With the passage retrieval task our conclusions are restricted to the BM25 weighting function with respect to retrieval effectiveness. However our conclusions with regard to retrieval efficiency are generic: we do not see *TermId* as being a viable partitioning method for any computationally intensive passage retrieval method because of communication requirements.

• Our conclusions with respect to the routing/filtering task are restricted to hillclimbers for term selection in retrieval efficiency, but generic with respect to retrieval effectiveness. Examining more of the search space only appears to yield a different maximim.

The generic statements we made here with respect to retrieval efficiency also apply the synthetic models we have derived for the tasks, thereby strengthening the models.

11.3 REFLECTIONS ON THE SYNTHETIC MODELS

How successful were our synthetic models in being able to distinguish between the distribution methods, the relative performance difference between them and form generic statements about the performance of parallel IR systems? Our main stated aim, that of producing models which could differentiate between the partitioning methods was successful on all tasks under discussion in this thesis. In particular, the prediction that a sequential sort would be a bottleneck for probabilistic search on *TermId* partitioning was validated. Furthermore, many of the assumptions made in the models such as those for load imbalance were reasonable approximations.

However, all models are simplifications in some sense and are therefore likely to be weak in some areas. It is clear from the examination of all tasks that our modelling of communication lead to problems in being able to distinguish the relative performance on distribution methods. An example of the problems encountered was that the synthetic models for indexing predicted that the gap between *DocId* and *TermId* builds would decrease, while our experiments showed that difference actually increased. Similarly our synthetic probabilistic model performance predicted that search *TermId* with a parallel sort would be nearer *DocId*, while our experiments show they were nearer *TermId* with sequential sort. These errors occurred partly due the high bandwidth network assumption used and partly due to naive communication modelling.

There was however, one positive result from this failure: as on-the-fly distribution was poor for the routing/filtering task it is clear that *TermId* partitioning, which would require more communication, would not be a viable method. Another interesting failure was on the passage retrieval task when trying to predict the performance of short queries on the distributed method: it is hard to see how any model could deal with non-determinism of this type. With respect to any generic statements we do not see how we can say any more beyond what we have stated in the previous section: however, the models are functional and therefore adaptable to the modelling of different techniques. We would hope that this work on performance modelling could be taken further and turned into an analytical model which would then be able to predict actual performance of many IR tasks using various distribution methods.

11.4 CHOOSING AN APPROACH

We have seen the motivations for using parallelism in IR and some of the methods which have been used in chapter 2. Our work in this thesis has taken this information further and we have demonstrated that the *DocId* partitioning method is the most effective distribution method for inverted file data on most tasks. In the context of the information given we describe a rationale for choosing a parallel IR approach. We assume that one or more of the reasons described in chapter 2 exists for choosing parallel IR systems in the first instance.

The central issue behind choosing an approach is that of index maintenance, in particular of the insertion rate compared with the query rate (Stone, 1987). A further issue is that of index generation: Hawking (1991) pointed out that building indexes for inverted files with the size of 8192 Gigabytes would take so long that the document retrieved would only ever be of historic interest (however it should be noted that processor speeds improved and memory sizes have grown a lot since 1991). We therefore suggest some criteria for choosing either an approach described in chapter 2 or one described in this thesis whichever is appropriate. We provide empirical evidence from this thesis to back these assertions up where required.

If normal keyword searches are required with no update we would recommend the use of inverted files partitioning using *DocId*. We have demonstrated speed advantage using this method for both probabilistic search and passage retrieval tasks. If search types such as regular expressions are required, then the pattern match method would be the most suitable. Regular expressions are difficult to implement on signature and inverted file methods and would be restricted in the two-phase search.

It is possible that even with a high level of parallelism, there exists an update rate which could not be handled by our index update technique due to restrictions on resources such as buffer space. In such a case we would suggest that the two-phase search or vector processing methods be used. Insertion of documents is much cheaper than inverted files in the chosen methods and queries are therefore much less likely to be affected by delays engendered by insertion. Where other types of document maintenance are required such as document alteration or deletion, or all three maintenance operations, then the use of two-phase search or vector methods would be preferred. Block deletions are not an issue since they are relatively inexpensive.

However where the query rate exceeds the insertion rate the use of *DocId* inverted files is recommended. We have demonstrated that reduction in the cost of maintaining indexes is possible by using parallelism (see chapter 8) and our method may be able offer much faster access to documents than would normally be possible with inverted file indexes. For very large databases it is possible to reduce downtime by using parallelism to insert documents in batches and increase system availability. Where the availability of documents is not such an important issue, batch updates would be preferred.

What of the other methods described in chapter 2 such as clustering and connectionist approaches? Because of the extra computation needed we would only recommend their use if some gain in retrieval effectiveness was found, using empirical experiment based on users relevance judgements. In some cases it is hard to justify the use of some methods, such as the application of parallel relational databases to IR: the use of parallel relational databases does not bring benefits in terms of retrieval effectiveness or efficiency.

11.5 FURTHER RESEARCH

A number of very important issues in Parallel IR have yet to be addressed. Some have been identified in this thesis; others were described in MacFarlane et al (1997). These include concurrent transaction service on inverted files, extended Boolean models, connectionist approaches, load balancing methods for term selection, other combinatorial optimisation methods and further work on synthetic modelling for parallel IR.

11.5.1 *Concurrent transaction service*

One of the most important areas to investigate is the issue of using parallelism to improve the performance of concurrent transaction service on inverted files. We have found some evidence in the examination of the probabilistic search and index update tasks that *TermId* partitioning could be useful in this context. We found that the workload over a number of queries was spread fairly evenly, with only a small level of imbalance recorded. This evidence in itself is not enough however, since it is possible that transactions may conflict over the same data residing on the same nodes creating hotspots. There have been some conflicting opinions on what would be the best method for concurrent transaction service. Lu and McKinley (1999) argue that partial collection replication improves both performance and scalability for large scale distributed IR systems, while Ribeiro-Neto et al (1999) argue for the use of a *TermId* partitioning method together with various compression techniques. It would be useful to do a comparison using these techniques with the *DocId* method.

To be able to service updates and multiple-queries simultaneously, an effective concurrency control mechanism may be required. In order to complete such research we need an evaluation methodology, as the ones we have currently are not adequate for the task. Once we have such a methodology we would be able to study the use of parallel computing for efficient update on inverted files and concurrency control mechanisms on the inverted file to prevent loss of retrieval efficiency and effectiveness (MacFarlane et al, 1996).

11.5.2 *Extended Boolean models*

To the best of our knowledge, no work has been done on applying parallel computing to extended Boolean models such as MMM, P-NORM and Paice (Fox et al, 1992). The MMM and Paice models use fuzzy set theory, while the P-NORM model uses a distance based theory. The models have been shown to produce better results than ordinary Boolean systems at the cost of extra computation. In the case of P-NORM this computational cost is very large. Work in the area of applying parallel techniques to these models is merited. These extended Boolean models provide the possibility of an increase in retrieval effectiveness: therefore evaluation of these methods in terms of precision and recall is regarded as essential.

11.5.3 *Connectionist models of IR*

Rasmussen (1992) identified the need for more work in the area of connectionist approaches, pointing out that there had been very little work at that point in the intersection between network models in IR and parallel computing for network models outside of IR. To the best of our knowledge there has been little further progress in the area, and Rasmussen's statement still holds true.

11.5.4 *Load balancing for term selection algorithms*

It is clear from the evidence found in the examination of the replication parallelism results that work on load balancing for the Find Best and CSP algorithms could be useful. Figs 11-1 and 11-2 show load imbalance figures both for the number of evaluations and number of skipped words (that is words dropped from the evaluation process after 4 failures, see chapter 10, section 10.2) made on the TREC8 runs. We take the load imbalance (LI) metric defined in chapter 3, sub-section 3.4.1.8 and apply it to evaluation and word skip data.

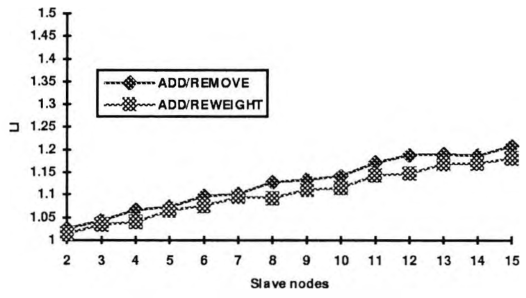


Fig 11-1. Imbalance in evaluations for TREC8 routing experiments

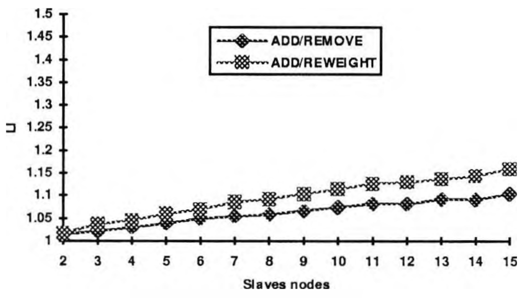


Fig 11-2. Imbalance in skipped words for TREC8 routing experiments

It can be seen from the diagrams that the trend of imbalance is upward which is consistent with the trend found when examining imbalance in time on slaves nodes in chapter

10, section 10.4.1. There is clearly a correlation between increasing imbalance on time, evaluations and word skips, but as LI is much higher on time than it is on evaluations and skip data therefore the interaction between the imbalances is complicated. The clue to solving the problem is establishing what this relationship is, and a method of using such data to balance load. We do not believe that a simple exchange of words would necessarily work as evaluations do not have a uniform cost: what is done on the evaluation is as important as the number of evaluations done.

Some form of dynamic re-distribution of terms amongst nodes needs to be formulated. A natural point for any re-distribution would be at the end of one iteration. Any method of load balancing should not increase overheads such that any advantage gained by balancing load is lost. Another important factor that needs to be taken into consideration is that any re-distribution technique may well have an effect on the course of term selection itself for some algorithms, e.g. CFP/CSP. This may affect the final choice of terms for the query to be applied to the test set, and by implication the final result. There is no guarantee that one method of load balancing for one algorithm/operation pair would be useful for another, but it would be nice to derive a generic term re-distribution method.

One of the algorithms, namely CFP will be problematic when the issue of term re-distribution is considered. The very nature of the choose first policy is that is it possible for one node to find a good term immediately where at least one other has to inspect all of its terms. Therefore there is an inbuilt load balancing problem with the method. We could consider stopping a given iteration each time a good term was found, but that would require a further complicated and expensive communication interactions both to find a positive term and communicate this information to all the nodes. Further research could be useful, however.

11.5.5 *Other methods for combinatorial optimisation*

We could consider the use of machine learning (Hutchinson, 1994) tabu search (Glover and Laguna, 1997) and pattern recognition (Kitter, 1986) techniques in order to optimise routing/filtering queries. A great deal of research into search space methods has been done in machine learning using methods such as genetic algorithms and neural networks that are both very computationally intensive processes. Tabu search is a meta-heuristic which can be used to manage other heuristics in order to examine parts of the search space which would not normally be looked at with a single search strategy. Some of the selection algorithms used in pattern recognition are similar to the hillclimbers used in this study (Kitter, 1986), particularly Find Best with *add only* and *remove only* operations. We could therefore treat the query optimisation discussed in this research as a pattern recognition problem, treating different combinations of the query as a pattern. The problem would be to find the best yield pattern in the query. Parallelism could be used to speed up these methods, providing they are able to show a retrieval effectiveness benefit on the test set.

11.5.6 *Further work on the synthetic model*

Further work on the synthetic modelling technique used in this thesis is merited both in terms of strengthening the actual model and extending it for use in other tasks not studied here. The key aspect to concentrate on initially will be the modelling of communication, given that it was such a significant problem in our models. The problems in being able to do this successfully should not be underestimated. The model will not only have to cope with interactions between two processors, it will also have to model the pattern of communication throughout the whole system (one of the reasons for our simplified modelling of communication was to avoid this complexity). An interesting and worthwhile piece of research would be to extend the functional modelling and use formal techniques to prove various aspects of it: this may well provide insights that would not be obtained otherwise. When many problems have been solved by using synthetic modelling, and more of an understanding of the theory of performance of parallelism in IR is obtained, it may well be possible to derive an analytical model of performance. Hopefully, this analytical model would be able to predict the actual performance in a given IR task, more accurately than we can at present.

11.6 IMPLICATIONS FOR THE FIELD

What are the implications for the field of parallelism in IR or IR general for the findings of this thesis? The first is that *TermId* partitioning does not seem have much of a future and in most cases any researcher looking to use parallelism to speed up their IR algorithms need look no further than the *DocId* partitioning method. The second is that we have demonstrated that it is possible to use a synthetic model to compare the performance of different algorithms on a given task, without the need for an analytical model. This simplifies the process of modelling considerably, giving a researcher the tools they need to choose a distribution method avoiding the need for complex modelling. It may be possible to extend the synthetic modelling technique to more analytical methods, but this will always be at the cost of extra complexity.

A very important issue is how well the algorithms and data structures used in this thesis will scale to hundreds or even thousands of processors in order to improve retrieval efficiency. The author does not think that for most of the collections such parallel machine sizes would be practical (except perhaps for the VLC2/WT100g collection - but there would be restrictions on even that). It is important to remember that there are collections current being used (notably by the web search engines) which are far bigger than the ones we have used, and we believe using such levels of parallelism are in fact useful. Gustafson's law predicts that as you increase data size, you also gain from parallelism (Hwang, 1993): either with a fixed size parallel machine (scalability) or varying machine size (scaleup), We believe that the algorithms and structures used for this thesis will scale up to data of the size of web data, with a few minor changes (for example it is highly likely that some form of query optimisation will be need to handle such large data sets).

With respect to retrieval effectiveness we have shown that just throwing extra computational resources at a problem will not lead to any gain in say precision/recall and a researcher who wishes to do such a thing may well be wasting their time. A researcher would be far better off deriving an algorithm, method or model which provides better retrieval effectiveness than the ones we have at present and may be computationally intensive. The use of parallelism can be then be considered. Parallelism is an enabling technology which allows us to tackle difficult problems, and is not an end in itself.

References

AALBERSBERG, I.J, and SIJSTERMANS, F. (1990)
InfoGuide: A full-text document retrieval system.
In: TJOA, A.M., and WAGNER, R., eds. Proceedings of the international conference of database and expert systems applications, DEXA'90. (Berlin: Springer-Verlag): 12-21.

ALASDAIR, R., BRUCE, A., MILLS, J.G., and SMITH, A.G. (1994)
CHIMP/MPI User Guide version 1.2, 22 June 1994.
EPCC-KTP-CHIMP-V2-USER1.2. Edinburgh Parallel Computing Centre.

ANU. (1994)
MPI user's guide.
ANU/Fujitsu CAP Research Program, Department of Computer Science, Australian National University.

BAILEY, P. and HAWKING, D. (1996).
A parallel architecture for query processing over a terabyte of text.
Technical Report TR-CS-96-04, Department of Computer Science, Canberra: Australian National University.

BALE, A.G., LITT, J. and PAVELIN, J. (1990).
The AMT DAP 500 system.
In: FOUNTAIN, T.J. and SHUTE, M.J., eds. Multiprocessor Computer Architectures. (Amsterdam: Elsevier Science Publishers B.V. North-Holland):155-184.

BEAULIEU, M.M., GATFORD, M., HUANG, X., ROBERTSON, S.E., WALKER, S and WILLIAMS, P. (1997).
Okapi at TREC-5.
In: VOORHEES, E.M. and HARMAN, D.K. eds, Proceedings of the Fifth Text Retrieval Conference (TREC-5), U.S.A, November 1996, SP 500-238 Gaithersburg.(Gaithersburg: NIST): 143-166.

BELL, G. (1992).
Ultracomputers: a teraflop before its time,
Communications of the ACM 35 (8): 27-47.

BLAIR, B.C. and MARON, M.E. (1985).
An evaluation of retrieval effectiveness for a full-text document retrieval system.
Communications of the ACM, 28 (3): 289-299.

BLAIR, B.C. and MARON, M.E. (1990).
Full-text information retrieval: further analysis and clarification.
Information Processing & Management, 26 (3): 437-447.

BOWLER, K.C., KENWAY, R.D., PAWLEY, G.S. ROWETH, D. and WILSON,G.V. (1989). An introduction to Occam-2 programming: 2nd Edition, (Lund: Chartwell-Bratt).

BROWN, E.W., CALLAN, J.P., CROFT, W.B., and MOSS, J.E.B. (1994).
Suporting full-text information retrieval with a persistent object store,
In: JARKE, M., BUBENKO, J. and JEFFERY, K. ed, Proceedings of EDBT'94, March 1994, LNCS 779, (Berlin:Springer-Verlag): 365-377.

CALLAN, J.P. (1994).
Passage-level evidence in document retrieval.
In: CROFT, W.B., and VAN RIJSBERGEN, C.J., eds, Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval, Dublin, July 1994, SIGIR'94, (London: Springer Verlag): 303-310

CARDENAS, A.F. (1975).
Analysis and performance of inverted data base structures.
Communications of the ACM, 18 (5): 253-263.

CARROLL, D.M., POGUE, C.A., and WILLETT, P. (1988).
Bibliographic pattern matching using the ICL Distributed Array Processor.
Journal of the American Society for Information Science, 39 (6): 390-399.

CLARKE, C.L.A., and CORMACK, G.V. (1995).
Dynamic inverted indexes for a distributed full-text retrieval system.
MultiText Project Technical Report MT-95-01. Department of Computer Science, Ontario: University of Waterloo.

CLARKE, C.L.A., CORMACK, G.V., and PALMER, C.R. (1998).
An overview of MultiText.
SIGIR Forum 32. (2): 14-15.

COLOURIS, G., DOLLIMORE, J. and KINDBERG, T. (1994). Distributed systems: concepts and design, Second Edition, (Wokingham:Addison-Wesley).

CORMACK, G.V., CLARKE, C.L.A., PALMER, C.R., TO, S.S.L., (1998).
Passage-based refinement (MultiText experiments for TREC-6).
In: VOORHEES, E.M., and HARMAN, D.K. ed. *Proceedings of the Sixth Text Retrieval Conference, Gaithersburg, U.S.A, November 1997*. SP 500-240. (Gaithersburg: NIST): 303-320.

COWIE, A.P., Ed. (1989). Oxford advanced learner's dictionary of current english, fourth edition, (Oxford: Oxford University Press).

CRINGEAN, J.K, MANSON, G.A., WILLETT, P., and WILSON, G.A. (1988).
Efficiency of text scanning in bibliographic databases using microprocessor-based multiprocessor networks.
Journal of Information Science. 14(6): 335-345.

CRINGEAN, J.K, LYNCH, M.F., MANSON, G.A., WILLETT, P., and WILSON, G.A. (1989).
Parallel processing techniques for information retrieval. Searching of textual and chemical databases using transputer networks.
Online Information 89. (Oxford: Learned Information): 447-452.

CRINGEAN, J.K, ENGLAND, R. MANSON, G.A. and WILLETT, P. (1990).
Parallel text searching in serial files uing a pocessor farm.
In: VIDICK, J.L, ed, Proceedings of the 13th International Conference on Research and Development in Information Retrieval. (New York: ACM Press): 429-453.

CRINGEAN, J.K, ENGLAND, R. MANSON, G.A. and WILLETT, P. (1991a).
Network design for the implementation of text searching using a multicomputer.
Information Processing & Management, 27 (4): 265-283.

CRINGEAN, J.K, ENGLAND, R. MANSON, G.A. and WILLETT, P. (1991b).
Nearest-neighbour searching in files of text signatures using transputer networks.
Electronic Publishing, 4 (4):185-203.

GROPP, W. and LUSK, E. (1998)
Users guide for MPICH, a portable Implementation of MPI.
Mathematics and Computer Science Division, Argonne National Laboratory, University of Chicago.

DATE, C.J. (1983). An introduction to database systems, Volume II, (Massachusetts: Addison-Wesley).

DEERWESTER, S.C., ZIFF, D.A., and WACLENA, K. (1990).
An architecture for full text retrieval systems.
In: TJOA, A.M., and WAGNER, R., eds. Proceedings of the international conference of database and expert systems applications, DEXA'90. (Berlin: Springer-Verlag): 22-29.

DEERWESTER, S., DUMAIS, S.T., FURNAS, G.W., LANDAUER,. T.K. and HARSHMAN, R. (1990)
Indexing by latent semantic analysis.
JASIS 41 (6):391-407.

DEITEL, H. M. (1990).Operating systems. 2nd edition, (Massachusetts: Addison-Wesley).

DEWITT, D., and GRAY. J. (1992).
Parallel database systems: the future of high performance database systems.
Communications of the ACM, 35 (6): 85-98.

DIXON, W. (2000). Personal communication.

DONGARRA, J.J., OTTO, S.W., SNIR, M., and WALKER, D. (1996).
A message passing standard for MPP and workstations.
Communications of the ACM, 39 (7): 84-90.

EFRAIMIDIS, P. GLYMIDAKIS, C. MAMALIS, B. SPIRAKIS, P. and TAMPAKAS, B. (1995).
Parallel text retrieval on a high performance supercomputer using the vector space model.
In: FOX, E.A., INGWERSEN, P and FIDEL, R. eds. Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Special Issue of SIGIR forum. (New York: ACM Press): 58-66.

FALOUTSOS, C. (1985).
Access methods for text.
ACM Computing Surveys, 17 (1): 49-74.

FEDOROWICZ, J. (1987).
Database performance evaluation in an indexed file enviroment.
ACM Transactions on Database Systems, 12 (1):85-110.

FLYNN, M. J. (1972).
Some computer organisations and their effectiveness.
IEEE Transactions on Computers, 21 (9): 948-960.

FOX, C. (1990)
A stop list for general text,
SIGIR FORUM, 24 (4): 19-35.

FOX, E., BETRABET, S., KOUSHIK, M., and LEE, W. (1992).
Extended boolean models.
In: FRAKES, W.B. and BAEZA-YATES, R., eds. Information Retrieval. Data Structures and Algorithms. (N.J.: Prentice-Hall): 393-418.

FRAKES, W.B. (1992).
Introduction to information storage and retrieval systems.
In: FRAKES, W.B. and BAEZA-YATES, R., eds. Information Retrieval. Data Structures and Algorithms. (N.J.: Prentice-Hall): 1-12.

FRIEDER, O., and SEIGELMANN, H.T. (1993)
On the allocation of documents in multiprocessor information retrieval systems.
In: KORFHAGE, R, RASMUSSEN, E.M., and WILLETT, P., eds, Proceedings of Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. (New York: ACM Press): 230-239.

FUJITSU. (1994).
AP1000 user guide.
Fujitsu Laboratories Ltd. Release 1.4 March 1994,

GLOVER, F. and LAGUNA, M. (1997). Tabu search. (Boston: Kluwer Academic Publishers).

GONNET, G.H., BAEZA-YATES, R.A., and SNIDER, T. (1992).
New indices for text: PAT trees and PAT arrays.
In: FRAKES, W.B. and BAEZA-YATES, R., eds. Information Retrieval. Data Structures and Algorithms. (N.J.: Prentice-Hall): 66-82.

GRANDI, F., TIBERIO, P. and ZEZULA, P. (1992).
Frame-sliced partitioned parallel signature files.
In: BELKIN, N.J., INGWERSEN, P., and PEJTERSEN, A.M., eds. Proceedings of the 15th annual conference on research and development in Information Retrieval, SIGIR'92. (New York: ACM Press): 286- 297.

GROSSMAN, D.A., HOLMES, D.O., and FRIEDER, O. (1995).
A parallel DBMS approach to IR in TREC-3.
In: HARMAN, D.K., ed. Proceedings of Third Text Retrieval Conference, Gaithersburg, USA, November 1994, SP 500-226. (Gaithersburg: NIST): 279-288.

GROSSMAN, D.A., HOLMES, D.O., FRIEDER, O., NGUYEN, M.D. and KINGSBURY, C.E. (1996).
Improving accuracy and run-time performance for TREC-4.
In: HARMAN, D.K., ed. Proceedings of Fourth Text Retrieval Conference, Gaithersburg, USA, November 1995, 500-236. (Gaithersburg: NIST): 433-442.

HARMAN, D.K.(1992).
Relevance feedback and other query modification techniques.
In: FRAKES, W.B, and BAEZA-YATES, R., eds. Information Retrieval, Data Structures and Algorithms. (N.J.: Prentice-Hall): 241-263.

HARMAN, D.K. (1996).
Overview of the fourth text retrieval conference (TREC-4),
In: HARMAN, D.K. ed, Proceedings of the Fourth Text Retrieval Conference (TREC-4), Gaithersburg, U.S.A, November 1995, SP 500-236. (Gaithersburg: NIST): 1-24.

HARMAN, D., FOX, E., BAEZA-YATES, R., and LEE, W. (1992).
Inverted files.
In: FRAKES, W.B, and BAEZA-YATES, R., eds. Information Retrieval, Data Structures and Algorithms. (N.J.: Prentice-Hall): 28-43.

HAWKING, D. (1991).
High speed search of large text base on the fujitsu cellular array processor.
In: GUPTA, G., and PRITCHARD, P., eds. Proceedings of 4th Australian Supercomputer Conference, Bond University, December 1991. (Gold Coast: Bond University): 83-90.

HAWKING, D. (1992).
PADDY's progress (further experiments in free-text retrieval on the AP1000).
In: ISHII, M., ed. Proceedings of the 1st Parallel Computing Workshop, Kawasaki, Japan, November 1992. (Kawasaki: Fujitsu Parallel Computing, Research Facility): ANU-8.

HAWKING, D. (1994).

PADRE - A parallel document retrieval engine.

In: ISHII, M., ed. Proceedings of the 3rd Parallel Computing Workshop, Kawasaki, Japan, November 1994. (Kawasaki: Fujitsu Parallel Computing, Research Facility): P2-C.

HAWKING, D. (1995).

The design and implementation of a parallel document retrieval engine.

Technical Report TR-CS-95-08, Department of Computer Science. Canberra: Australian National University.

HAWKING, D. (1996).

Document retrieval performance on parallel systems.

In: ARABNIAL, H.R., ed. Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications, Sunnyvale, California, August 1996. (Athens: CSREA): 1354-1365.

HAWKING, D. (1998).

Efficiency/effectiveness trade-offs in query processing.

SIGIR Forum, 32 (2): 16-22.

HAWKING, D. and BAILEY, P. (1993).

Towards a practical information retrieval system for the fujitsu AP1000.

In: ISHII, M., ed. Proceedings of the 2nd Parallel Computing Workshop, Kawasaki, Japan, November 1993. (Kawasaki: Fujitsu Parallel Computing, Research Facility): P1-S.

HAWKING, D. and BAILEY, P. (1995).

PADRE user manual.

Department of Computer Science, (Canberra: Australian National University).

HAWKING, D., BAILEY, P., CAMPBELL, D., THISTLEWAITE, P. and TRIDGELL, A. (1995).

A PADRE in MUFTI (a multi user free text retrieval intermediary).

In: DARLINGTON, J., ed. Proceedings of the 4th Parallel Computing Workshop, Imperial College, London, September 1995. (London: Imperial College / Fujitsu Parallel Computing Research Facility): 75-84.

HAWKING, D., CRASWELL, N. and THISTLEWAITE, P. (1999).
Overview of TREC-7 very large collection track.
In: VOORHEES, E.M. and HARMAN, D.K., eds. Proceedings of Seventh Text Retrieval Conference (TREC-7), Gaithersburg, USA, November 1998. NIST SP 500-242, (Gaithersburg: NIST): 257-268.

HAWKING, D. and THISTLEWAITE, P. (1995).
Searching for meaning with the help of a PADRE.
In: HARMAN, D.K., ed. Proceedings of Third Text Retrieval Conference, Gaithersburg, USA, November 1994. SP 500-226, (Gaithersburg: NIST): 257-268

HAWKING, D. and THISTLEWAITE, P. (1996).
Proximity operators - so near and yet so far.
In: HARMAN, D.K., ed. Proceedings of Fourth Text Retrieval Conference, Gaithersburg, USA, November 1995. SP 500-236 (Gaithersburg: NIST): 131-144.

HAWKING, D. and THISTLEWAITE, P. (1998).
Overview of TREC-6 very large collection track.
In: VOORHEES, E.M. and HARMAN, D.K., ed. Proceedings of Seventh Text Retrieval Conference (TREC-7), Gaithersburg, USA, November 1997, SP 500-242, (Gaithersburg: NIST): 93-106.

HAWKING, D., THISTLEWAITE, P. and CRASWELL, N. (1998).
ANU/ACSys TREC-6 experiments.
In: VOORHEES, E.M. and HARMAN, D.K., ed. Proceedings of Seventh Text Retrieval Conference (TREC-7), Gaithersburg, USA, November 1997, SP 500-242 (Gaithersburg: NIST): 275-290.

HAWKING, D., VOORHEES, E.M., CRASWELL, N. and BAILEY, P. (2000).
Overview of TREC-8 web track.
In: VOORHEES, E.M. ed. Proceedings of Eight Text Retrieval Conference (TREC-8), Gaithersburg, USA, November 1999, (Gaithersburg: NIST): to appear

HOCKNEY, R. W. and JESSHOPE, C.R. (1988). Parallel computing 2. (Bristol: IOP Publishing).

HOCKNEY, R.W. (1993).
Performance parameters and benchmarking of supercomputers.
In: DONGARRA, J.J., and GENTZSCH, W., Computer Benchmarks: Advance in Parallel Computers 8. (Amsterdam: North-Holland): 41-63.

HOLLAAR, L.A. (1991).
Special-purpose hardware for text searching: past experience, future potential. Information Processing & Management, 27 (4): 371-378.

HOLLAAR, L.A. (1992).
Special-purpose hardware for information retrieval.
In: FRAKES, W.B, and BAEZA-YATES, R., eds. Information Retrieval, Data Structures and Algorithms. (N.J.: Prentice-Hall): 443-458.

HULL, D.A (1999).
The TREC-7 filtering track: description and analysis.
In: VOORHEES, E.M. and HARMAN, D.K., eds. Proceedings of Seventh Text Retrieval Conference (TREC-7), Gaithersburg, USA, November 1998. NIST SP 500-242, (Gaithersburg: NIST): 33-56.

HULL, D.A. and ROBERTSON, S.E. (2000).
The TREC-8 filtering track final report.
In: VOORHEES, E.M. and HARMAN, D.K., eds. Proceedings of Eighth Text Retrieval Conference (TREC-8), Gaithersburg, USA, November 1999. NIST SP XXX-XXX, (Gaithersburg: NIST): to appear.

HUTCHINSON, A. (1994). Algorithm learning. (Oxford: Clarendon Press).

HURSON, A.R., MILLER, L.L., PAKZAD, S.H. and CHENG, J.B. (1990).
Specialized parallel architectures for textual databases.
In: YOVITS, M., ed. Advances In Computers. Vol. 30, Academic Press.: 1-37.

HWANG, K. (1993). Advanced Computer Architecture: Parallelism, Scalability, Programmability. (Singapore: McGraw-Hill).

JEONG, B., and OMIECINSKI, E. (1995).
Inverted file partitioning schemes in multiple disk systems.
IEEE Transactions on Parallel and Distributed Systems, 6 (2): 142-153.

KAPALEASWARAN, T.N., and RAJARAMAN, V. (1990).
Parallel search methods of a document database in a distributed computer system: a case study.
Journal of Information Science, 16: 291-298.

KASZKIEL, M., and ZOBEL, J. (1997).
Passage retrieval revisited.
In: BELKIN, N.J., NARASIMHALU, A.D. and WILLETT, P., Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Philadelphia, SIGIR'97, (New York: ACM Press): 178-185.

KIRSCH, S. (1998).
The future of Internet search: Infoseek's experiences searching the Internet.
SIGIR Forum, 32 (2): 3-7.

KIM, J.Y. (2000). Personal communication.

KITTER, J. (1986).
Feature selection and extraction.
In: YOUNG, T.Y. and FU, K., eds. Handbook of Pattern Recognition and Image Processing, New York:Academic Press): 59-83.

KNIGHT-RIDDER. (1997). 1997 Complete database catalogue, Knight-Ridder Information, Inc.

KWOK, K.L. (1989).
A neural network for probabilistic information retrieval.
In: BELKIN, N.J., and VAN RIJSBERGEN, C.J., eds. Proceedings of the 12th annual conference on research and development in Information Retrieval, SIGIR'89. (New York: ACM Press): 21-30.

KWOK, K.L., and GRUNFELD, L. (1994).
TREC2 document retrieval experiments using PIRCS.
In: HARMAN, D.K., ed. Proceedings of the Second Text Retrieval Conference, Gaithersburg, USA, November 1993, SP 500-215. (Gaithersburg: NIST): 233-242.

LAWRENCE, S. and GILES C.L. (1999)
Accessibility of information on the web.
Nature 400: 107-109.

LETSCHE, T.A. and BERRY, M.W. (1997)
Large-scale information retrieval with latent semantic indexing.
Information Sciences 100: 105-137.

LEWIS. D.D. (1996).
The TREC-4 filtering Track.
In: HARMAN, D.K., ed, Proceedings of the Fourth Text Retrieval Conference, Gaithersburg, U.S.A, November 1995, SP 500-236. (Gaithersburg: NIST): 165-180.

LEWIS. D.D. (1997a).
The TREC-5 filtering Track.
In: VOORHEES, E.M, and HARMAN, D.K., ed, Proceedings of the Fifth Text Retrieval Conference, Gaithersburg, U.S.A, November 1996, SP 500-238. (Gaithersburg: NIST 1997): 75-96.

LEWIS. D.D. (1997b).
Reuters-22178 text categorization test collection.
http://www.research.att.com/~lewis/reuters21578/README.txt

LINOFF, G., and STANFILL, C. (1993).
Compression of indexes with full positional information in very large text databases.
In: KORFHAGE, R, RASMUSSEN, E.M., and WILLETT, P., eds, Proceedings of Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. (New York: ACM Press): 88-95.

LOVINS, J.B. (1968)

Development of a stemming algorithm.

Mechanical Translation and Computational Liguistics, Vol. 11.

LU, Z. and MCKINLEY, K.S. (1999).

Partial replica selection based on relevance for information retrieval.

In: HEARST, M., GEY, F. and TONG, R., Proceedings of the 22nd International Conference on Research and Development in Information Retrieval, SIGIR'99, (New York: ACM Press): 97-104.

MACFARLANE, A., ROBERTSON, S.E., and MCCANN, J.A. (1996).

On concurrency control for inverted files.

In: JOHNSON, F.C., ed. Proceedings of the 18th BCS IRSG Annual Colloquium on Information Retrieval Research, March 26-27 1996, Manchester. (Manchester: BCS IRSG): 67-79.

MACFARLANE, A., ROBERTSON, S.E., and MCCANN, J.A. (1997).

Parallel computing in information retrieval - an updated review

Journal of Documentation, 53 (3): 274-315.

MACFARLANE, A., MCCANN, J.A. and ROBERTSON, S.E. (1999a).

PLIERS: a parallel information retrieval system using MPI.

In: DONGARRA, J., LUQUE, E. and MARGALEF, T., eds. Proceedings of 6th European PVM/MPI Users' Group Meeting, Barcelona, Lecture Notes in Computer Science 1697, (Berlin: Springer-Verlag): 317-324.

MACFARLANE, A., ROBERTSON, S.E., and MCCANN, J.A. (1999b).

PLIERS at VLC2.

In: VOORHEES, E.M. and HARMAN, D.K., eds. Proceedings of Seventh Text Retrieval Conference (TREC-7), Gaithersburg, USA, November 1998. SP 500-242, (Gaithersburg: NIST): 327-336

MACFARLANE, A., ROBERTSON, S.E., and MCCANN, J.A. (2000a).
PLIERS at TREC 8.
In: VOORHEES, E.M. and HARMAN, D.K., eds. Proceedings of Eight Text Retrieval Conference (TREC-8), Gaithersburg, USA, November 1999. SP 500-246, (Gaithersburg: NIST): 241-252.

MACFARLANE, A., MCCANN, J.A. and ROBERTSON, S.E. (2000b).
Parallel search using partitioned inverted files.
In: DE LA FUENTE, P., ed, Proceedings of String Processing and Information Retrieval - SPIRE 2000, September 2000, A Coruna, Spain. (Los Alamitos:IEEE Computer Society Press), 209-220.

MACLEOD, K.J. and ROBERTSON W. (1991).
A neural algorithm for document clustering.
Information Processing & Management, 27 (4): 337-46.

MASSAND, B., and STANFILL, C. (1994).
An information retrieval test-bed on the CM-5.
In: HARMAN, D.K., ed. Proceedings of Second Text Retrieval Conference, Gaithersburg, USA, November 1993. SP 500-215. (Gaithersburg: NIST): 117-122.

ODDY, R.N. and BALAKRISHNAN, B. (1991).
PTHOMAS: an adaptive information retrieval system on the connection machine.
Information Processing & Management, 27 (4): 317-335.

OZKARAHAN, E. (1991).
System architectures for information processing.
Information Processing & Management, 27 (4): 347-369.

PANAGOPOULOS, G. and FALOUTSOS, C. (1994).
Bit-sliced signature files for very large text databases on a parallel machine architecture.
In: MATTHIAS, J., BUBENKO, J., and JEFFERY, K., eds. Proceedings of EDBT'94. (Heidelberg: Springer-Verlag): 379-392.

POGUE, C.A. and WILLETT, P. (1987a)
Text searching algorithms for parallel processors.
British Library Research Paper 11, (London: British Library).

POGUE, C.A., and WILLETT, P. (1987b).
Use of text signatures for document retrieval in a highly parallel environment.
Parallel Computing, 4: 259-268.

POGUE, C.A., RASMUSSEN, E.M., and WILLETT, P. (1988).
Searching and clustering of databases using the ICL Distributed Array Processor.
Parallel Computing, 8: 399-407.

RASMUSSEN, E.M., and WILLETT, P. (1989).
Efficiency of hierarchic agglomerative clustering using the ICL Distributed Array Processor.
Journal of Documentation, 45 (1): 1-24.

RASMUSSEN, E.M. (1991).
Introduction: parallel processing and information retrieval.
Information Processing & Management, 27 (4): 225-263.

RASMUSSEN, E. M. (1992).
Parallel information processing.
In: WILLIAMS, M.E., Annual Review of Information Science and Technology (ARIST), Volume 27, (N.J.: American Society for Information Science): 99-130.

REDDAWAY, S.F. (1991).
High speed text retrieval from large databases on a massively parallel processor.
Information Processing & Management, 27 (4): 311-316.

RIBEIRO-NETO, B., MOURA, E.S., NEUBERT, M.S., and ZIVIANI, N. (1999).
Efficient distributed algorithms to build inverted files.
In: HEARST, M., GEY, F. and TONG, R., Proceedings for the 22nd International Conference on the Research and Development in Information Retrieval, SIGIR'99, (New York: ACM Press): 105-112.

ROBERTSON, S.E. (1990).

On term selection for query expansion, documentation note.

Journal of Documentation, 46, No 4: 359-364.

ROBERTSON, S.E., and SPARCK JONES, K. (1976)

Relevance weighting of search terms.

JASIS, May-June: 129-145.

ROBERTSON, S.E., and SPARCK JONES, K. (1994)

Simple, proven approaches to text retrieval.

Technical Report No. 356. University of Cambridge Computer Laboratory.

ROBERTSON, S.E. AND WALKER, S. (1995). On the logic of search sets and non-boolean retrieval. Unpublished paper.

ROBERTSON, S.E., WALKER, S., JONES, S., HANCOCK-BEAULIEU, M.M. and GATFORD, M. (1995).

Okapi at TREC-3.

In: HARMAN, D.K., ed. Proceedings of Third Text Retrieval Conference, Gaithersburg, USA, November 1994, NIST SP 500-226, (Gaithersburg: NIST): 109-126.

ROBERTSON, S.E., WALKER, S., JONES, S., HANCOCK-BEAULIEU, M.M., GATFORD, M. and PAYNE, A. (1996).

Okapi at TREC-4.

In: HARMAN, D.K., ed. Proceedings of the Fourth Text Retrieval Conference, Gaithersburg, U.S.A. November 1995, NIST SP 500-236, (Gaithersburg: NIST): 73-96.

RUNGSAWANG, A., TANGPONG, A., and LAOHAWEE, P. (1999).

Parallel DISR text retrieval system.

In: DONGARRA, J., LUQUE, E. and MARGALEF, T., eds. Proceedings of 6th European PVM/MPI Users' Group Meeting, Barcelona, Lecture Notes in Computer Science 1697, (Berlin: Springer-Verlag): 325-332.

SALTON, G., and BERGMARK, D. (1981).
Parallel computations in information retrieval.
In: HANDLER, W., ed. Proceedings of CONPAR'81. (Berlin: Springer-Verlag): 328-342.

SALTON, G. (1986).
Another look at automatic text-retrieval systems.
Communications of the ACM, 29 (7): 648-656.

SALTON, G., and BUCKLEY, C. (1988).
Parallel text search methods.
Communications of the ACM, 31 (2): 202-215.

SCHIETTECATTE, F. (1996). Personal communication.

SEIGELMANN, H.T., and FRIEDER, O. (1993).
Document allocation in multiprocessor information retrieval systems.
In: ADAM, N.R., and BHARGAVA, B.K. eds. Advanced Database Systems. (Berlin: Springer-Verlag): 289-310.

SHARMA, R. (1989).
A generic machine for parallel information retrieval.
Information Processing and Management, 25 (3): 223-235.

SHISSLER, J. (2000). Personal communication.

SHOENS, K., TOMASIC, A., and GARCIA-MOLINA H. (1994).
Synthetic workload performance analysis of incremental updates.
In: CROFT, W.B., and VAN RIJSBERGEN, C.J., eds. Proceedings of the 17th annual international ACM-SIGIR conference on research and development in Information Retrieval. SIGIR94. (London: Springer-Verlag): 329-338.

SINGHAL, A. (1998).
AT&T at TREC-6.
In: VOORHEES, E.M. and HARMAN, D.K., ed. *Proceedings of the Sixth Text Retrieval Conference, Gaithersburg, U.S.A, November 1997*, SP 500-240. (Gaithersburg: NIST): 215-226.

SILVERSTEIN, C., HENZINGER, M., MARAIS, H, and MORICZ, M. (1999).
Analysis of a very large web search engine log.
SIGIR Forum, 33 (1): 6-12.

SKILLICORN, D.B. (1995a).
Structured parallel computation in structured documents.
External Technical Report, Ontario: Queen's University, Canada..

SKILLICORN, D.B. (1995b)
A generalisation of indexing for parallel document search.
External Technical Report, Ontario: Queen's University, Canada.

STANFILL, C. and KAHLE, B. (1986).
Parallel free-text search on the connection machine system.
Communications of the ACM, 29 (12): 1229-1239.

STANFILL, C., THAU, R., and WALTZ, D. (1989).
A parallel Indexed algorithm for Information Retrieval.
In: BELKIN, N.J., and VAN RIJSBERGEN, C.J., eds. Proceedings of the 12th annual conference on research and development in Information Retrieval, SIGIR'89. (New York: ACM Press): 88-97.

STANFILL, C. (1990).
Partitioned posting files: a parallel inverted file structure for information retrieval.
In: VIDICK, J.L, ed, Proceedings of the 13th International Conference on Research and Development in Information Retrieval. (New York: ACM Press): 413-428.

STANFILL, C, and THAU, R. (1991).
Information retrieval on the connection machine: 1 to 8192 Gigabytes.
Information Processing & Management, 27 (4): 285-310.

STANFILL, C. (1992).
Parallel information retrieval algorithms.
In: FRAKES, W.B, and BAEZA-YATES, R., eds. Information Retrieval, Data Structures and Algorithms. (N.J.: Prentice-Hall): 413-428.

STEPHEN, G.A., and MATHER, P. (1994).
What is SP?
The Computer Journal, 37 (9): 745-752.

STEPHEN, G.A. (1994).
What is SP?: A Reply, Correspondence.
The Computer Journal, 38 (3): 255-256.

STEWART, M. and WILLETT, P. (1987).
Nearest neighbour searching in binary search trees: simulation of a multiprocessor system.
Journal of Documentation, 43 (2): 93-111.

STONE, H.S. (1987).
Parallel querying of large database: a case study.
IEEE Computer, 20 (10): 11-21.

SUNDERAM, V.S. 1990
PVM: A framework for parallel distributed computing.
Concurrency: Practice and Experience, 2 (4): 315-339.

TANENBAUM, A.S. (1990). Structured computer organisation. 3rd edition, (N.J.: Prentice-Hall).

TOMASIC, A., and GARCIA-MOLINA, H. (1992).
Performance of inverted indices in shared-nothing distributed text document information retrieval systems.
Technical Report STAN-CS-92-1434, Department of Computer Science, (C.A.: Stanford University.

TOMASIC, A., and GARCIA-MOLINA, H. (1993a)
Performance of inverted indices in shared-nothing distributed text document information retrieval systems.
Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems, (Los Alomitos: IEEE Computer society press): 8-17.

TOMASIC, A., and GARCIA-MOLINA, H. (1993b).
Caching and database scaling in distributed shared-nothing information retrieval systems.
In: BUNEMAN, P., and JAJODIA, S., eds, Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data. (N.Y.: ACM Press): 129-138.

TUSON, A. (1998).
Optimisation with hillclimbing on steriods: an overview of neightbourhood search techniques.
In: Proceedings of the 10th Young Operational Research Conference, Operational Research Society, 1998:141-156.

VOORHEES, E.M. and HARMAN, D.K. (1999).
Overview of the seventh text retrieval conference (TREC-7).
In: VOORHEES, E.M. and HARMAN, D.K., eds, Proceedings of Seventh Text Retrieval Conference (TREC-7), Gaithersburg, USA, November 1998, SP 500-242, (Gaithersburg: NIST): 1-23.

WALDEN, M., and SERE, K. (1989).
Free text retrieval on transputer networks.
Microprocessors and Microsystems, 13 (3): 179-187.

WALKER, S., ROBERTSON, S.E., and BOUGHANEM, M. (1998).
OKAPI at TREC-6.
In: VOORHEES, E.M. and HARMAN, D.K., ed, Proceedings of the Sixth Text Retrieval Conference, Gaithersburg, U.S.A, November 1997, SP 400-240, (Gaithersburg: NIST): 125-136.

WALTZ, D. (1987).
Applications of the connection machine.
IEEE Computer, 20 (1): 85-97.

WILLETT, P., and RASMUSSEN, E.M. (1990). Parallel Database Processing. (London: Pitman).

WILSON, E. (1996a).
Using hypertext and parallel processing to integrate multi-purpose, multi-structural databases.
Hypertext paper, Kent: University of Kent at Canterbury.

WILSON, E. (1996b)
Hypertext and parallel processing: browsing and retrieval.
Hypertext paper, Kent: University of Kent at Canterbury.

WIRTH, N. (1986). Algorithms & data structures. (N.J.: Prentice-Hall).

WOLFF, J.G. (1994a).
A scaleable technique for best-match retrieval of sequential information using metrics-guided search.
Journal of Information Science, 20 (1): 16-28.

WOLFF, J.G. (1994b).
What is SP?: A reply, correspondence.
The Computer Journal, 38 (3): 253-255.

WOLFRAM, D. (1992a).

Applying informetric characteristics of databases for ir system file design, part i:
informetic models.

Information Processing and Management, 28 (1): 121-133.

WOLFRAM, D. (1992b).

Applying informetric characteristics of databases for ir system file design, part ii:
simululation comparisions.

Information Processing and Management, 28 (1): 135-151.

YOUNT, R.J., VRIES, J.K., and COUNCILL, C.D. (1991).

The medical archival system: an information retrieval system base on distributed parallel
processing.

Information Processing & Management, 27 (4): 379-389.

GLOSSARY

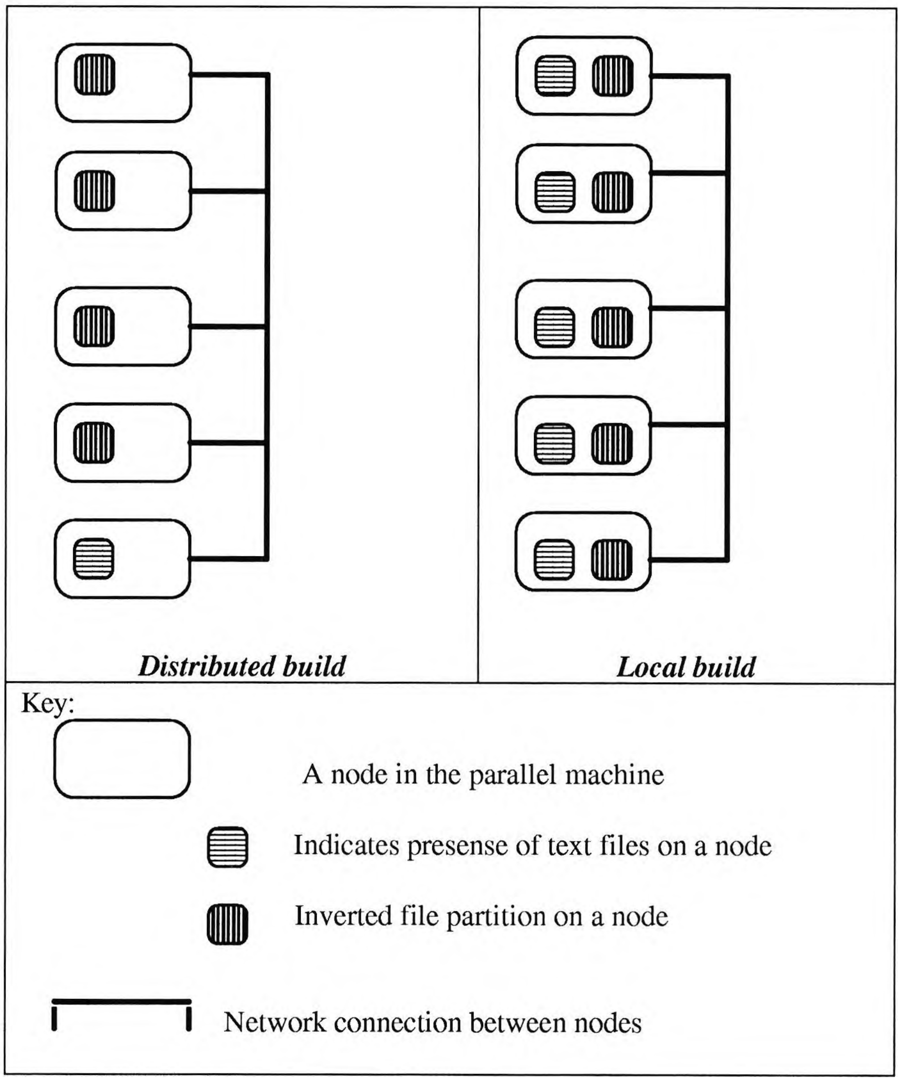
Add Only	A term set operation in which good terms are added to a query.
Add/Remove	A term set operation in which good terms are either added to a query or negative terms are removed from the query.
Add/Reweight	As add only, but set weights on the document identifier set of a term are varied.
Accumulated set	A document idenfifier set which is the result of all merges done in term selection.
Base set	Set of terms chosen from the top of the term pool: forms the initial query.
BM25	Best Match function number 25 (see chapter 3).
BSSF	Bit Sliced Signature File.
CAP	Choose All Positive: A Steepest-ascent hillclimber which accumulates the best terms using a given operation over all terms in one iteration.
CF allocation	Method of term allocation in *TermId* to partition using a collection frequency criterion.
CFP	Choose First Positive: A First-ascent hillclimber which selects the first good term using a given operation in one iteration.
CM-2	Thinking Machines Connection Machine 2.
CPU	Central Processing Unit.
CSP	Choose Some Positive: Parallel version of CAP which makes a selection on a subset of terms in one iteration.
DAP	Distributed Array Processor.
Distributed Build	Method of building indexes where text is distributed from a single node.
Distributed memory	Architecture in which memory is distributed amongst processors.
Distributed Passage Processing	Method of parallel passage retrieval which processes passages on a whole database level.
Document id set	Set of documents in which a given term occurs.
DocId	Parititioning method which assigns all document data for a given document to one index partition

DSM	Architecture in which memory is physically distributed, but logically shared amongst processors.
Efficiency	Measure of how well a parallel machine is used.
Elapsed time	Time to complete a task from start to finish. Can be average elapsed time.
Evaluation	Operation applied during term selection to a term using a given metric.
Evaluation set	The term pool minus the base set: term selection is done on these terms.
Farmer process	Process in Distributed Build indexing which distributes text to nodes.
FB	Find Best: A Steepest-ascent hillclimber which selects the best term using a given operation over all terms in one iteration.
FSA	Finite State Automata.
Gigabytes	2^{30} bytes.
Global Merge Process	Process which exchanges data between nodes in order to create a distributed *TermId* inverted file.
Granularity	Measure or size of individual computation in parallel computing.
Indexer process	Process in Local Build which analyses text and builds inverted file to the local disk.
IDF	Inverse Document Frequency.
Intra-query	Methods available within queries i.e. parallelism.
Inter-query	Methods available between queries i.e. parallelism.
Intermediate set	Result of a merge between a current terms document identifier set and the accumulated set during term selection: evaluations are applied to this set.
Inverted File	Index organisation of keywords and the documents they occur in.
I/O	Input / Output.
LI	Load Imbalance.
Local Build	Method of indexing where all processing is kept local to the node.
Local Passages Processing	Method of parallel passage retrieval which kepts passage processing local to a node.
Megabytes	2^{20} bytes.
Merge Costs	Percentage of time spent merging over all the processors.
MCU	Master Control Unit.

Mhz	Megahertz: processor clock speed.
MIMD	Multiple Instruction Multiple Data machine architecture.
MISD	Multiple Instruction Single Data machine architecture.
MMM Model	Fuzzy set based extended boolean model.
NEWS grid	North South East West interconnect for parallel architecture.
Paice Model	Fuzzy set based extended boolean model.
Partition	Fragment of Inverted file on a nodes disk.
PE	Processing Element.
P-NORM Model	Distance based extended boolean model.
Precision	Measure of relevant documents retrieved.
Precision Points	Number of relevant documents at 5,10,15 and 20 documents retrieved.
Process farm	A set of processes where a farmer process distributes work to worker processes.
Recall	Measure of retrieved relevant documents.
Regular Expressions	Used to search for a number of patterns rather than a single pattern.
Remove only	A term set operation in which negative terms are removed from the query.
Replication	Duplication of the inverted file on all nodes of a shared nothing parallel computer.
Scalability	A measure of how well the algorithm scales on the same equipment.
Scaleup	"The ability of an N-times larger system to perform an N-times larger job in the same elapsed time as the original system"
signature	Document surrogate of n bits, where terms are hashed to m bits.
SIMD	Single Instruction Multiple Data machine architecture.
SISD	Single Instruction Single Data machine architecture.
Shared everything	Architecture in which memory and disk are shared among processors.
Shared memory	Architecture in which memory is shared amongst processors.
Shared nothing	Architecture in which a processor has its own memory and disk.
SP	Theory of computing as compression, applied to pattern matching.
Speedup	Measure of speed advantage of parallelism over uniprocessors.
Streams	A sequence of instructions or data operated on by a CPU.
Surrogate coding	see signature.

TermId	Paritioning method which assigns all term data for a given term to one partition.
Term Pool	A term set created by using relevance feedback on documents marked relevant to the topic and applying a term selection value criterion to chose the top N terms.
Term Selection	Process of selecting terms for a routing/filtering query using an algorithm (i.e FB) together with a term operation (i.e. Add Only).
TF allocation	Method of term allocation in *TermId* to partition using a term frequency criterion.
Throughput	Number of discrete jobs done in a given time period.
Timing Process	Process which times local build indexing elapsed time.
TREC	Annual Text Retrieval Conference run by the National Institute of Standards and Technology in the United States.
TSV	Term Selection Value: Weight assigned to a term based on relevance feedback data.
VLDC	Variable Length Don't Care pattern match.
WC allocation	Method of term allocation in *TermId* to partition using a word count criterion.
Worker	Process which creates index data from raw text in Distributed Build indexing.
Zipf distribution	Distribution which suggests that a few words will occur in many documents, while many words will occur in few documents.

Appendix A1: Fig A1-1. Examples of build methods for distributed inverted files


Distributed build
Local build
Key:
A node in the parallel machine
Indicates presense of text files on a node
Inverted file partition on a node
Network connection between nodes

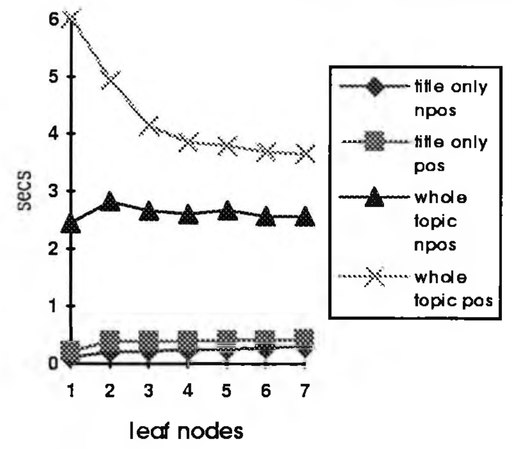


Fig A2-1. BASE1 [*TermId*]: search average elapsed time in seconds (sequential sort: CF distribution)

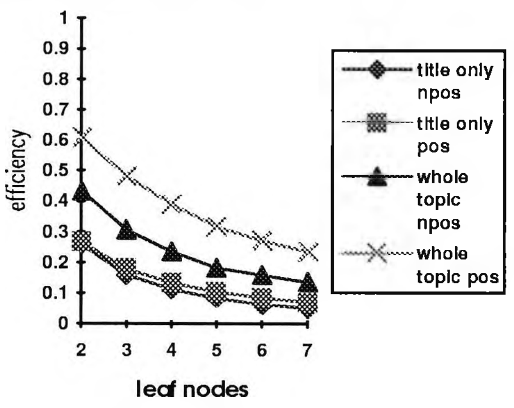


Fig A2-3. BASE1 [*TermId*]: search parallel efficiency (sequential sort: CF distribution)

Leaf nodes	Title Only		WholeTopic	
-	NPOS	POS	NPOS	POS
2	63%	40%	55%	34%
3	69%	48%	63%	44%
4	73%	53%	69%	54%
5	75%	55%	72%	57%
6	78%	58%	75%	60%
7	80%	61%	73%	62%

Table A2-1. BASE1 [*TermId*]: search overheads in % of total time (sequential sort: CF distribution)

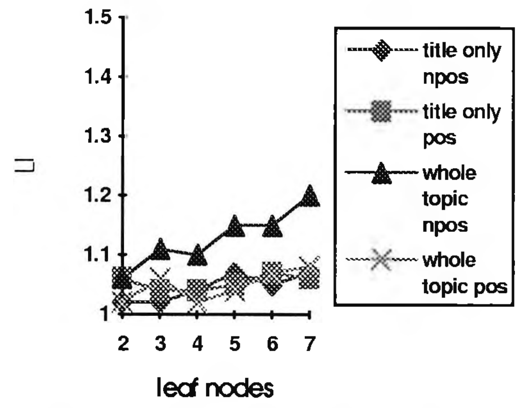


Fig A2-4. BASE1 [*TermId*]: search load imbalance (sequential sort: CF distribution)

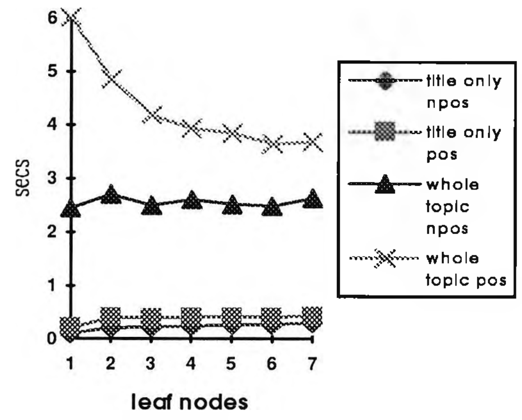


Fig A2-5. BASE1 [*TermId*]: search average elapsed time in seconds (sequential sort: TF distribution)

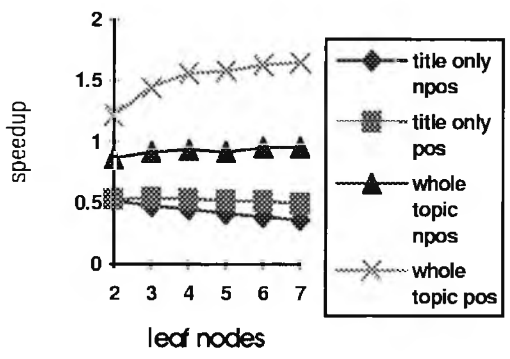


Fig A2-2. BASE1 [*TermId*]: search speedup (sequential sort: CF distribution)

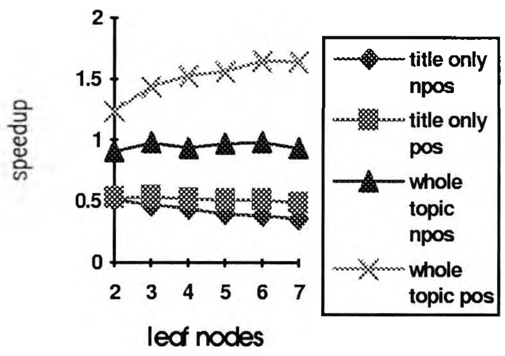


Fig A2-6. BASE1 [*TermId*]: search speedup (sequential sort: TF distribution)

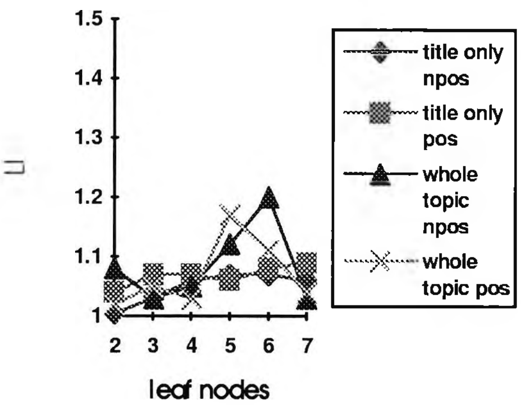


Fig A2-8. BASE1 [*TermId*]: search load imbalance (sequential sort: TF distribution)

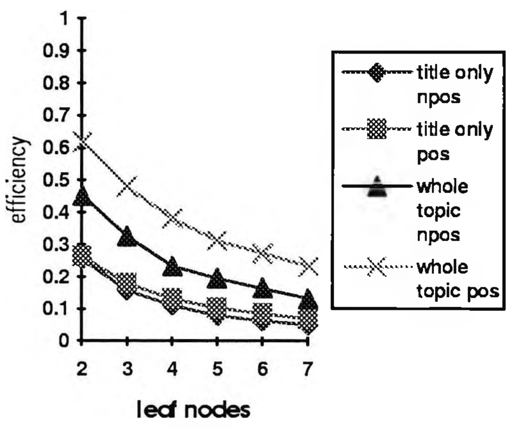


Fig A2-7. BASE1 [*TermId*]: search parallel efficiency (sequential sort: TF distribution)

Leaf nodes	**Title Only**		**WholeTopic**	
-	NPOS	POS	NPOS	POS
2	64%	41%	53%	34%
3	68%	47%	67%	44%
4	72%	52%	74%	55%
5	71%	56%	73%	54%
6	77%	57%	76%	58%
7	79%	59%	81%	64%

Table A2-2. BASE1 [*TermId*]: search overheads in % of total time (sequential sort: TF distribution)

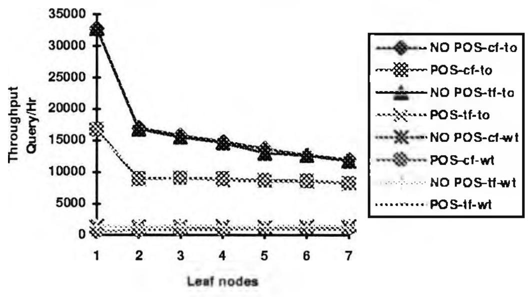


Fig A2-9. BASE1 [*TermId*]: search throughput in queries/hour (CF and TF distributions)

Appendix A3. Further Retrieval Efficiency Results for *on-the-fly* distribution: Routing/Filtering task

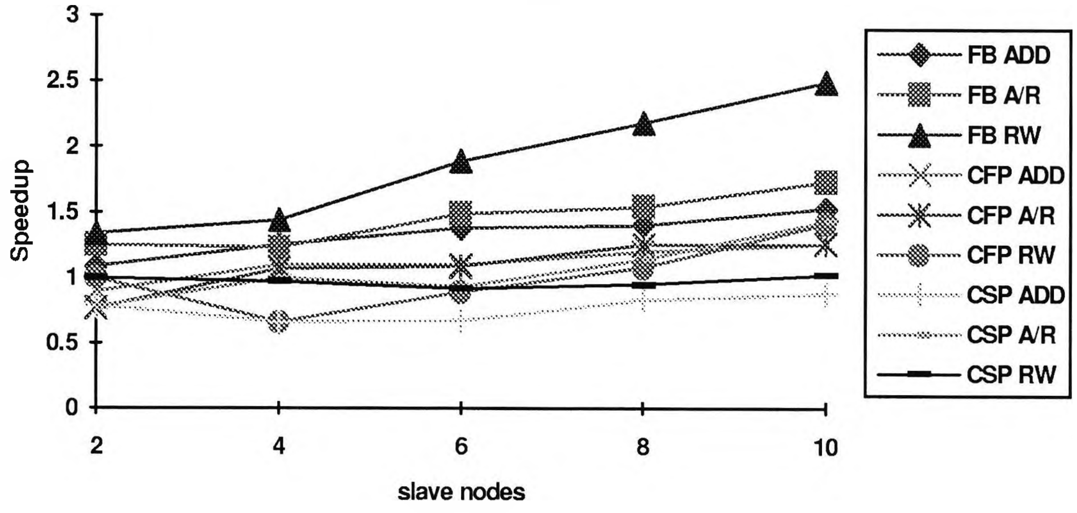


Fig A3-1. ZIFF-DAVIS [*On-the-fly*]: speedup for term selection algorithms (Network)

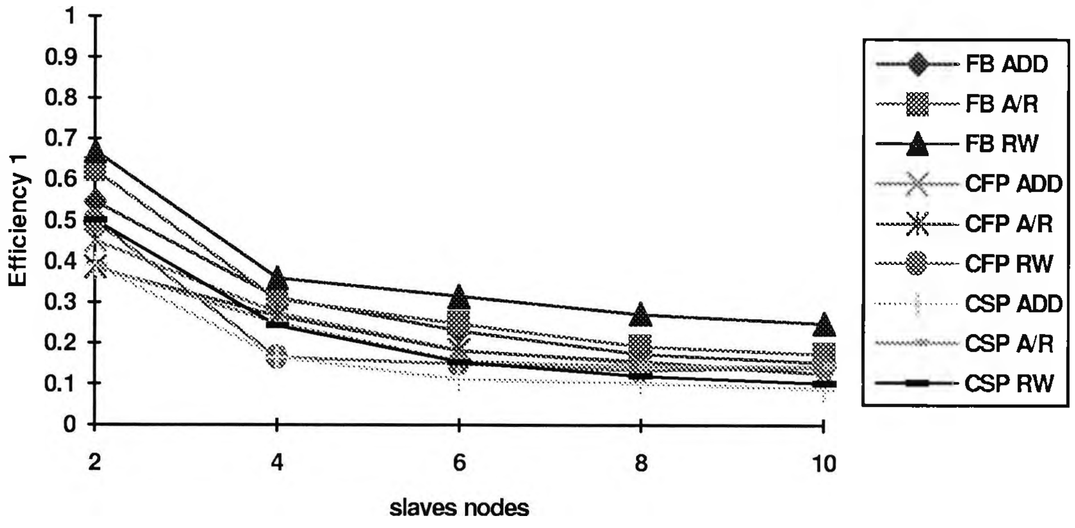


Fig A3-2. ZIFF-DAVIS [*On-the-fly*]: parallel efficiency for term selection algorithms (Network)

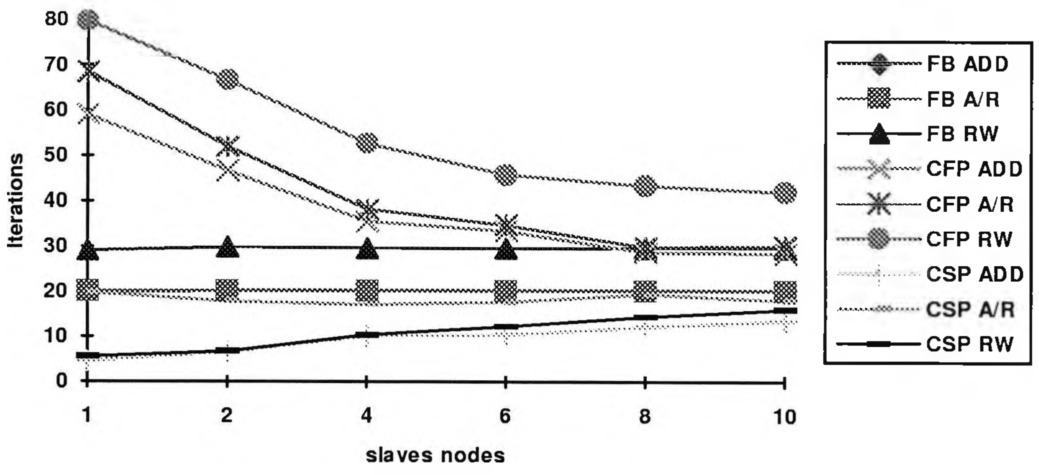


Fig A3-3. ZIFF-DAVIS [*On-the-fly*]: outer iterations to service term selection (Network)

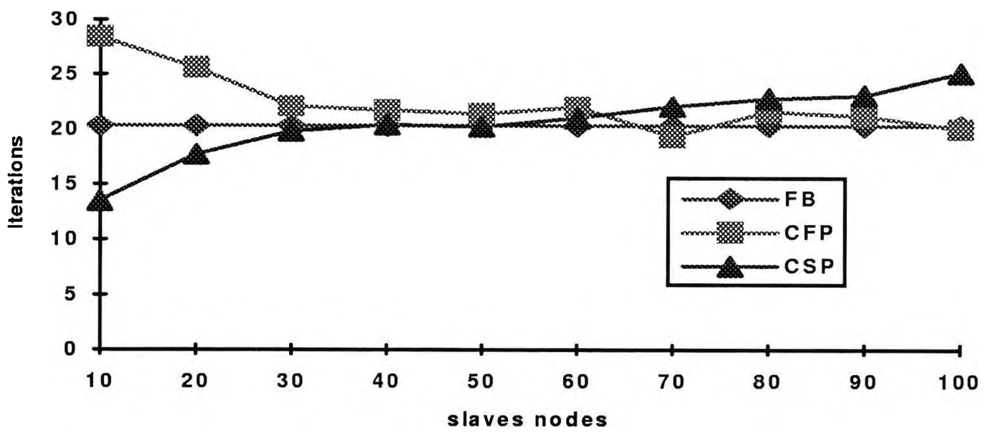


Fig A3-4. ZIFF-DAVIS [*On-the-fly*]: outer iterations to service term selection (AP1000)

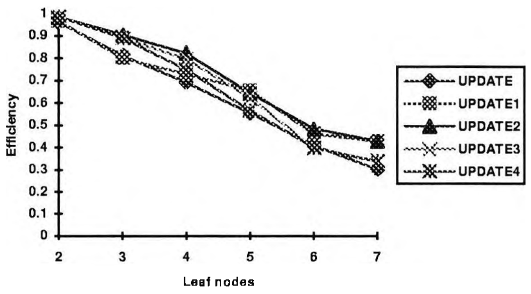


Fig A4-1. BASE1 [*DocId*]: parallel efficiency for update transactions (postings only)

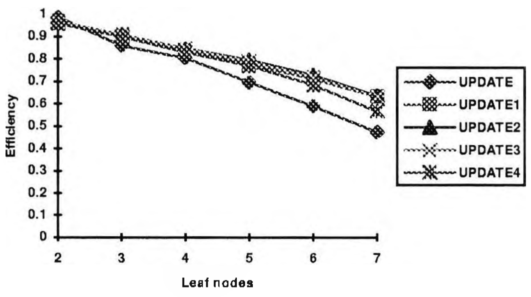


Fig A4-3. BASE1 [*DocId*]: parallel efficiency for update transactions (position data)

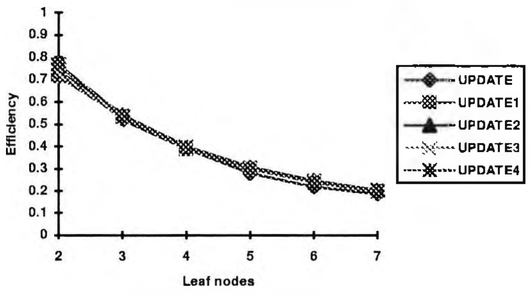


Fig A4-2. BASE1 [*TermId*]: parallel efficiency for update transactions (postings only)

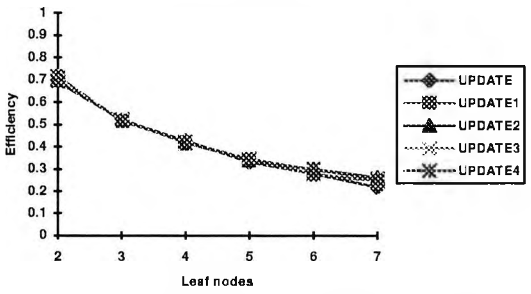


Fig A4-4. BASE1 [*TermId*]: parallel efficiency for update transactions (position data)

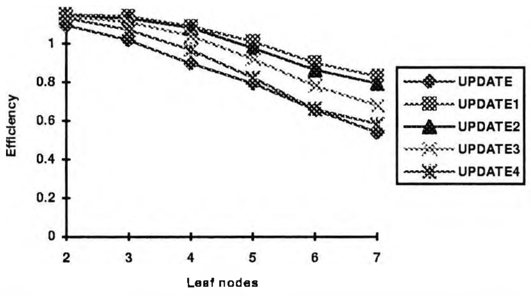


Fig A4-5. BASE1 [*DocId*]: parallel efficiency for all transactions (postings only)

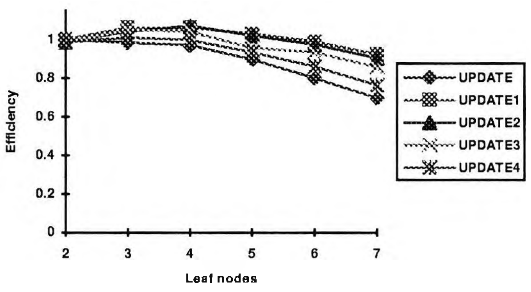


Fig A4-7. BASE1 [*DocId*]: parallel efficiency for all transactions (position data)

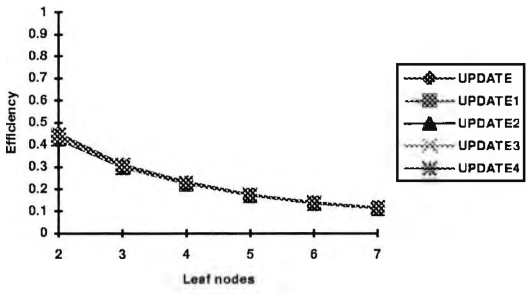


Fig A4-6. BASE1 [*TermId*]: parallel efficiency for all transactions (postings only)

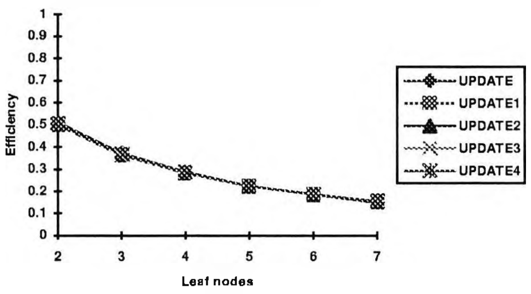


Fig A4-8. BASE1 [*TermId*]: parallel efficiency for all transactions (position data)

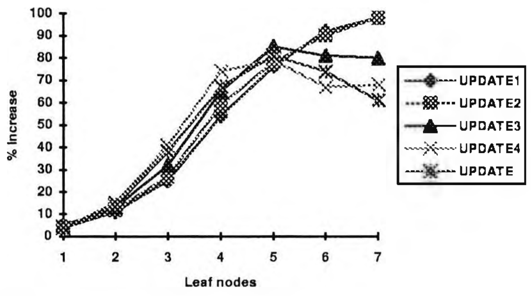


Fig A4-9. BASE1 [*DocId*]: % increase from normal average transaction elapsed time during index update (postings only)

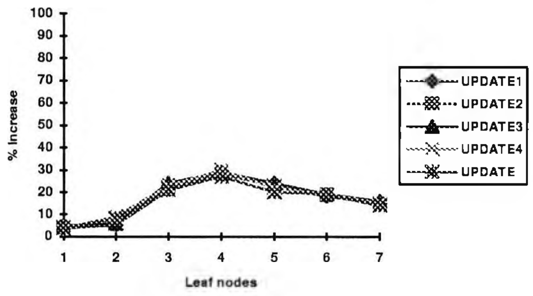


Fig A4-11. BASE1 [*TermId*]: % increase from normal average transaction elapsed time during index update (postings only)

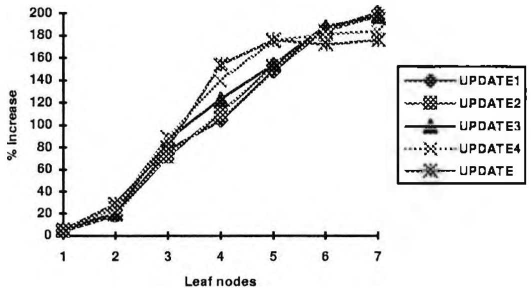


Fig A4-10. BASE1 [*DocId*]: % increase from normal average transaction elapsed time during index update (position data)

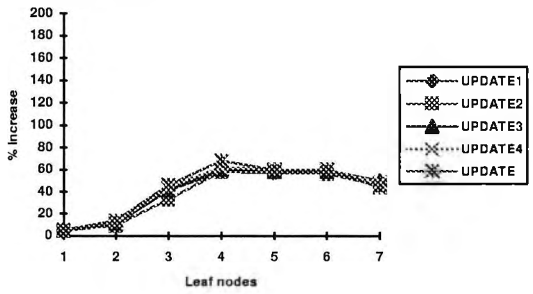


Fig A4-12. BASE1 [*TermId*]: % increase from normal average transaction elapsed time during index update (position data)

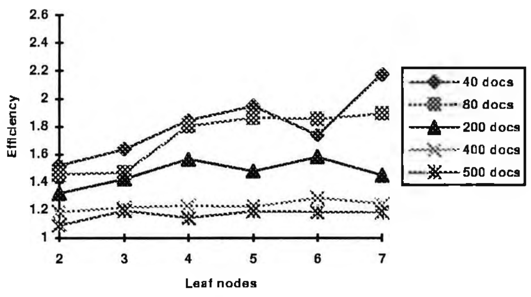


Fig A4-13. BASE1 [*DocId*]: Parallel efficiency for index reorganisation (postings only)

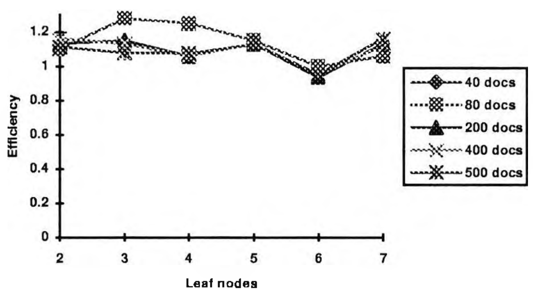


Fig A4-15. BASE1 [*TermId*]: Parallel efficiency for index reorganisation (postings only)

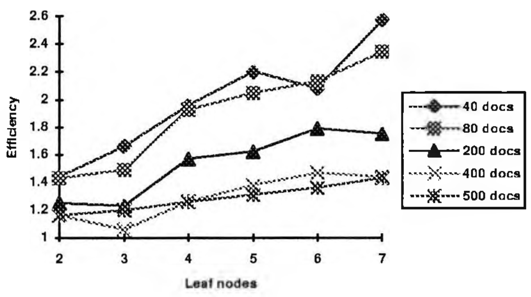


Fig A4-14. BASE1 [*DocId*]: Parallel efficiency for index reorganisation (position data)

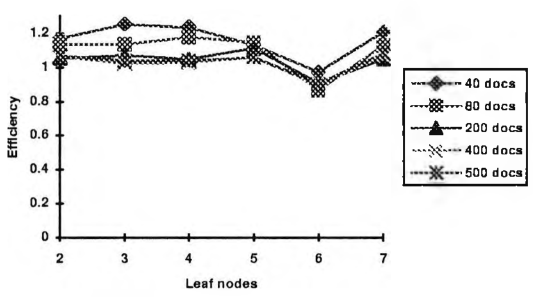


Fig A4-16. BASE1 [*TermId*]: Parallel efficiency for index reorganisation (position data)

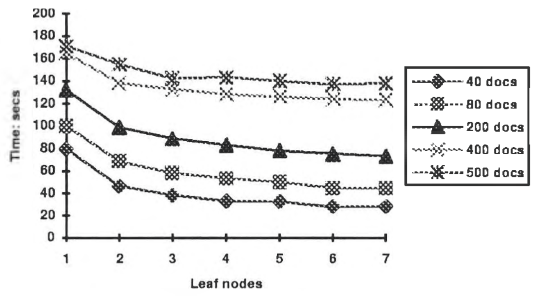


Fig A4-17. BASE1 [*DocId*]: Accumulated total time for index reorganisation (postings only)

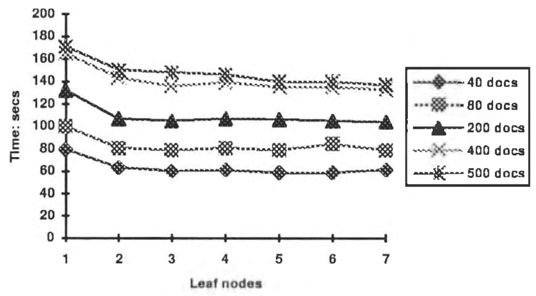


Fig A4-19. BASE1 [*TermId*]: Accumulated total time for index reorganisation (postings only)

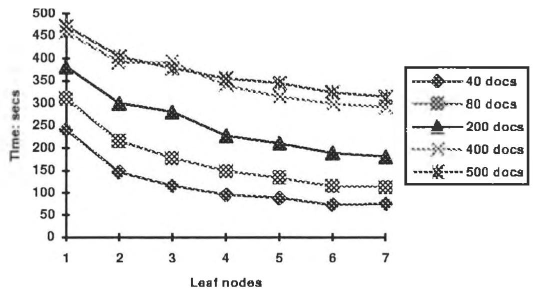


Fig A4-18. BASE1 [*DocId*]: Accumulated total time for index reorganisation (position data)

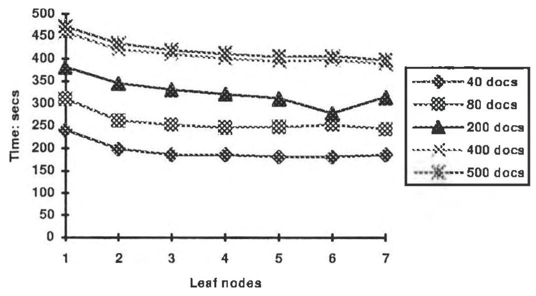


Fig A4-20. BASE1 [*TermId*]: Accumulated total time for index reorganisation (position data)

Metric	**Collection**	**40 Docs**	**80 Docs**	**200 Docs**	**400 Docs**	**500 Docs**
Position Data						
Total Time secs	BASE1 BASE10	65.9 677	105 1,053	174 1,644	280 2,466	312 2,608
Scalability on Total Time secs	BASE10	.973	.993	1.06	1.13	1.20
Postings Only (No Positions)						
Total Time (secs)	BASE1 BASE10	26.6 235	42.5 372	72.8 590	123 907	280 800
Scalability on Total Time (secs)	BASE10	1.13	1.14	1.23	1.36	1.60

Table A4-1. BASE1/BASE10[*DocId*]: scalability on index reorganisation

Appendix A5 - Synthetic models chapter appendix.

P	LI[P]
2	1.02
3	1.03
4	1.05
5	1.06
6	1.08
7	1.09
8	1.11
9	1.13
10	1.14
20	1.16
30	1.18
40	1.20
50	1.21
60	1.23
70	1.25
80	1.27
90	1.29
100	1.31

Table A5-1. Load imbalance estimates for LI[P] variable

A5.1 SEQUENTIAL MODEL FOR INDEXING

A5.1.1 *Analyse Documents*

Strip words from documents:	$d * n$
Insert Word Into Block:	$\log(n)$

$$dn\,log(n) * T_{cpu}$$

A5.1.2 *Save Intermediate Results*

Number of intermediate saves:	$(dn/ BSIZE)$
Cost per intermediate save	$(BSIZE * T_{i/o})$

$$(dn/ BSIZE) * (BSIZE * T_{i/o})$$

A5.1.3 *Merge Phase*

Load Blocks:	$(dn/ BSIZE) * (BSIZE * T_{i/o})$
Write Blocks:	$(dn/ BSIZE) * (BSIZE * T_{i/o})$
Merge Blocks:	$(dn/ BSIZE) * (BSIZE * T_{cpu})$

$$2((dn/ BSIZE) * (BSIZE * T_{i/o})) + (dn/ BSIZE) * (BSIZE * T_{cpu})$$

A5.1.4 *Sequential Indexing Model*

Combing the equations declared in sections A5.1.1 to A5.1.3 gives us the following sequential synthetic indexing model;

$$INDEX_{seq}(d,n,BSIZE) = dn\,log(n) * T_{cpu} + 3((dn/ BSIZE) * (BSIZE * T_{i/o})) + (dn/ BSIZE) * (BSIZE * T_{cpu})$$

A5.2 PARALLEL MODELS FOR INDEXING

A5.2.1 *Distributing Documents to nodes*

$$dn/f * T_{comm}$$

A5.2.2 *Global Merge Phase*

$$(dn/ BSIZE) * (P * T_{comm})$$

A5.2.3 *DocId Indexing Models*

Using the function defined in section A5.1.4 and the equation defined in section A5.2.1, we can define synthetic models for *DocId* indexing. With distributed build ($INDEX_{Distr\_DocId}$) we also add the distribution component for text data (equation from section A5.2.1)

$$INDEX_{Local\_Docid}(d,n,P,BSIZE) = (INDEX_{seq}(d,n,BSIZE)/P) * LI[P]$$

$$INDEX_{Distr\_DocId}(d,n,f,P,BSIZE) = ((INDEX_{seq}(d,n,BSIZE)/P) * LI[P]) + (dn/f * T_{comm})$$

A5.2.4 *TermId Indexing Model*

The *distributed build TermId* model ($INDEX_{Distr\_TermId}$) must redefine one aspect of the sequential indexing model defined in section A5.1.4. The merge component defined in section A5.1.3 is doubled for the *TermId* model and the extra communication costs from section A5.2.2 are added. The revised index computation component is divided by the number of processors and multiplied by the load imbalance estimate.

$$\text{INDEX}_{\text{Distr\_TermId}}(d,n,f,P,BSIZE) = dn/f * T_{comm} + (dn/BSIZE) * (P * T_{comm}) + \frac{(dn\,log(n) * T_{cpu} + 6((dn/BSIZE) * (BSIZE * T_{i/o})) + 2(dn/BSIZE) * (BSIZE * T_{cpu}))}{P} * LI[P]$$

A5.3 SEQUENTIAL MODEL FOR PROBABILISTIC SEARCH

The sequential model for probabilistic search is made up the the following:

Load q Keyword sets:	$Load\_kw_{seq}(q,s) = q * T_{i/o}[s]$
Weight q Keyword sets:	$Weight\_kw_{seq}(q,s) = s*q * T_{cpu}$
Merge q-1 Keyword sets:	$Merge\_kw_{seq}(q,s) = (q-1)*(s+s) * T_{cpu}$
Sort final results set	$Sort\_set_{seq}(q,s) = R[q,s]\,log(R[q,s]) * T_{cpu}$

Put together these functions make up the synthetic search model for sequential probabilistic search;

$$SEARCH_{seq}(s,q) = Load\_kw_{seq}(q,s) + Weight\_kw_{seq}(q,s) + Merge\_kw_{seq}(q,s) + Sort\_set_{seq}(q,s)$$

A5.4 PARALLEL MODELS FOR PROBABILISTIC SEARCH

A5.4.1 *DocId Partitioning*

The parallel model using *DocId* partitioning for probabilistic search is made up the the following:

Communications Costs for *DocId*:	$Comms\_Search_{docid}(P) = 3(P* T_{comm})$
Send P requests for terms frequency:	$P* T_{comm}$
Send P Queries (with term frequency):	$P* T_{comm}$
Gather results from P for set size s:	$P* T_{comm}$
Load q Keyword sets:	$Load\_kw_{docid}(q,s,P) = q * T_{i/o}[s/P]$
Weight q Keyword sets:	$Weight\_kw_{par}(q,s,P) = (Weight\_kw_{seq}(q,s)/P) * LI[P]$
Merge q-1 Keyword sets:	$Merge\_kw_{par}(q,s,P) = (Merge\_kw_{seq}(q,s)/P) * LI[P]$
Sort final results set:	$Sort\_set_{par}(q,s,P) = ((R[q,s]/P)\,log(R[q,s]/P) * T_{cpu}) * LI[P]$

The *DocId* partitioning synthetic search model is therefore;

$$\mathrm{SEARCH}_{docid}(s,q,P) = \mathrm{Comms\_Search}_{docid}(P) + \mathrm{Load\_kw}_{docid}(q,s,P) + \mathrm{Weight\_kw}_{par}(q,s,P) + \mathrm{Merge\_kw}_{par}(q,s,P) + \mathrm{Sort\_set}_{par}(q,s,P)$$

A5.4.2 *TermId Partitioning 1 - Sequential Sort*

The parallel model using *TermId* partitioning for probabilistic search using a sequential sort is made up the the following:

Communications Costs

for *TermId1*: $\mathrm{Comms\_Search}_{termid1}(s,q,P,SSIZE) = (R[s,q]/SSIZE)* P* T_{comm})+(P* T_{comm})$

Send p Queries (with term frequency):	$P* T_{comm}$
Gather results from p for set size s:	$(R[s,q]/SSIZE)* P* T_{comm}$
Load q Keyword sets:	$\mathrm{Load\_kw}_{termid}(q,s,P) = \lceil q/P \rceil * T_{i/o}[s/P[q]]$
Weight q Keyword sets:	$\mathrm{Weight\_kw}_{par}(q,s,P[q])$
Merge q-1 Keyword sets:	$\mathrm{Merge\_kw}_{par}(q,s,P[q])$
Sort final results set:	$\mathrm{Sort\_set}_{seq}(q,s)$

The *TermId* partitioning synthetic search model with sequential sort is therefore;

$$\mathrm{SEARCH}_{termid1}(s,q,P,SSIZE) = \mathrm{Comms\_Search}_{termid1}(s,q,P,SSIZE) + \mathrm{Load\_kw}_{termid}(q,s,P) + \mathrm{Weight\_kw}_{par}(q,s, P[q]) + \mathrm{Merge\_kw}_{par}(q,s,P[q]) + \mathrm{Sort\_set}_{seq}(q,s)$$

A5.4.3 *TermId Partitioning 2 - Parallel Sort*

The parallel model using *TermId* partitioning for probabilistic search using a parallel sort is made up the the following:

Communications Costs

for *TermId2*: $\text{Comms\_Search}_{termid2}(s,q,P,SSIZE) = 3(R[s,q]/SSIZE)* P* T_{comm})+(P* T_{comm})$

Send p Queries (with term frequency):	$P* T_{comm}$
Gather results from p for set size s:	$3(R[s,q]/SSIZE)* P* T_{comm}$
Load q Keyword sets:	$\text{Load\_kw}_{termid}(q,s,P)$
Weight q Keyword sets:	$\text{Weight\_kw}_{par}(q,s,P[q])$
Merge q-1 Keyword sets:	$\text{Merge\_kw}_{par}(q,s,P[q])$
Sort final results set:	$\text{Sort\_set}_{par}(q,s,P)$

The *TermId* partitioning synthetic search model with parallel sort is therefore;

$$\text{SEARCH}_{termid2}(s,q,P,SSIZE) = \text{Comms\_Search}_{termid2}(s,q,P,SSIZE) + \text{Load\_kw}_{termid}(q,s,P) + \text{Weight\_kw}_{par}(q,s,P[q]) + \text{Merge\_kw}_{par}(q,s,P[q]) + \text{Sort\_set}_{par}(q,s,P)$$

A5.5 SEQUENTIAL MODEL PASSAGE RETRIEVAL

Service q terms on PR documents each with (a(a-1))/2 inspected passages:

$$\text{Compute\_Pass}(PR,q,a) = T_{cpu} * PR * q*((a(a-1))/2)$$

A sort on the top PR documents is required to re-rank the final results set, cost is $T_{cpu}PR\,log(PR)$.

$$\text{PASSAGE}_{seq}(s,q,a,PR) = \text{SEARCH}_{seq}(s,q) + \text{Compute\_Pass}(PR,q,a) + T_{cpu}PR\,log(PR)$$

A5.6 PARALLEL MODELS FOR PASSAGE RETRIEVAL

A5.6.1 *DocId Models*

The *DocId* method simply applies P processors to the *Compute_Pass* computation defined in section A5.5:

$$\text{Compute\_Pass}_{par}(PR,q,a,P) = (\text{Compute\_Pass}(PR,q,a)/P) * LI[P]$$

The local passage processing cost model is constructed by simply adding the probabilistic *DocId* cost model from section A5.4.1 to the $\textit{Compute\_Pass}_{par}$ model;

$$\text{PASSAGE}_{docid\_local}(s,q,a,PR,P) = \text{SEARCH}_{docid}(s,q,P) + \text{Compute\_Pass}_{par}(PR,q,a,P)$$

The distributed passage retrieval method must also gather up data from nodes in order to choose the best PR documents in the collection. This requires four stages;

i) Gather the data from an initial probabilistic search (the top PR documents)

ii) Scatter this full set to the processors

iii) Gather up the full set from all the processors

iv) Do a final rank on the top PR documents

The estimate for this overhead is therefore:

i) Gather data (PR/SSIZE)/P * P:	PR/SSIZE (eliminated P)
ii) Scatter PR elements to P processors:	$T_{comm}(PR/SSIZE)*P$
iii) Gather PR elements from P processors:	$T_{comm}(PR/SSIZE)*P$
iv) Sort PR elements to obtain final rank:	$T_{cpu}PR\,log(PR)$

The model for overheads on distributed passage processing is therefore;

$$\text{OVERHEAD}_{pass}(PR,P,SSIZE) = T_{comm}((2(PR/SSIZE)*P)+ PR/SSIZE) + T_{cpu}PR\,log(PR)$$

The *DocId* distributed passage processing cost model is constructed by adding the probabilistic *DocId* cost model from section A5.4.1 to the *Compute_Pass*$_{par}$ model together with the OVERHEAD$_{pass}$ cost model;

$$\text{PASSAGE}_{docid\_distr}(s,q,a,PR,SSIZE,P) = \text{SEARCH}_{docid}(s,q,P) + \text{Compute\_Pass}_{par}(PR,q,a,P) + \text{OVERHEAD}_{pass}(PR,P,SSIZE)$$

A5.6.2 *TermId Models*

In *TermId* we must communicate the data for (a(a-1))/2 passages for PR documents on P processors:

$$\text{OVERHEAD}_{passtid}(a,PR,P) = T_{comm}(\ PR*P*((a(a-1))/2)$$

The *TermId* distributed passage processing cost models are constructed by adding the probabilistic *TermId* cost model from sections A5.4.2 and A5.4.3 to the *Compute_Pass*$_{par}$ model together with the *OVERHEAD*$_{pass}$ and *OVERHEAD*$_{passtid}$ cost models;

$$\text{PASSAGE}_{termid1}(s,q,a,PR,SSIZE,P) =$$

$SEARCH_{termid1}(s,q,P,SSIZE) + Compute\_Pass_{par}(PR,q,a,P[q]) + OVERHEAD_{pass}(PR,P,SSIZE) + OVERHEAD_{passtid}(a,PR,P)$

$PASSAGE_{termid2}(s,q,a,PR,SSIZE,P) =$
$SEARCH_{termid2}(s,q,P,SSIZE) + Compute\_Pass_{par}(PR,q,a,P[q]) + OVERHEAD_{pass}(PR,P,SSIZE) + OVERHEAD_{passtid}(a,PR,P)$

5.7 SEQUENTIAL MODELS FOR TERM SELECTION

A5.7.1 *Evaluation*

The cost of evaluation is broken down into the following;

Merge set for term with accumulated set:	$T_{cpu}*(s+s)$
Merge relevance judgements with temporary set:	$T_{cpu}*(s+r)$
Rank the temporary set using a sort:	$T_{cpu}*(R[q,s]\,log(R[q,s]))$

Put together these equations form the model for the cost of a single evaluation;

$EVAL(q,s,r) = T_{cpu}*((s+s) + R[q,s]\,log(R[q,s])+ (s+r))$

A5.7.2 *Total number of evaluations*

Maximum number of evaluations for the *find best* algorithm is:

$q*i$

Not all Keywords are inspected in i iterations:

$i* (i+1) * 0.5$

After each iteration one less term is inspected. This formula accumulates the total number of keywords not inspected in i iterations, as one term is always chosen. Put together with an estimate of the total number of terms skipped the function for inspected terms is:

$INSPECTED(q,i) = (qi - (i(i+1)0.5) - u(qi -(i(i+1)0.5)))$

A5.7.3 *Load costs for keywords*

The cost of loading term data is as follows;

Load q terms from disk each with set size s:	$q * T_{i/o}[s]$
Weight q terms each with set size s:	$q * s * T_{cpu}$

Putting these equations together yields the following load cost;

$LOAD(q,s) = q(T_{i/o}[s] +s*T_{cpu})$

A5.7.4 *Sequential Models for Term Selection*

Using the models defined in sections A5.7.1 to A5.7.3 we can now define the sequential cost models for term selection. For *add only* operation (ROUTING_{seq}) this is a simple process of multiplying the evaluation cost (see section A5.7.1) with the number of terms inspected (see section A5.7.2) and with an addition of load costs (see section A5.7.3). The model for add reweight (ROUTING_{seqw}) is constructed by factoring the total evaluation cost by the reweight variable w.

$$\text{ROUTING}_{seq}(s,r,i,q) = (\text{INSPECTED}(q,i) * \text{EVAL}(q,s,r)) + \text{LOAD}(q,s)$$

$$\text{ROUTING}_{seqw}(s,r,i,q,w) = (\text{INSPECTED}(q,i) * \text{EVAL}(q,s,r) * w) + \text{LOAD}(q,s)$$

A5.8 PARALLEL MODELS FOR TERM SELECTION

Basic term selection models with no synchronisation or communication costs is as follows;

$$\text{ROUTING}_{par}(s,r,i,q,P) = \frac{\text{ROUTING}_{seq}(s,r,i,q) * \text{LI}[P]}{P}$$

$$\text{ROUTING}_{parw}(s,r,i,q,P,w) = \frac{\text{ROUTING}_{seqw}(s,r,i,q,w) * \text{LI}[P]}{P}$$

5.8.1 *DocId Models*

The cost model for intra-set parallelism is;

Merge set costs:	$\text{Merge\_Route}_{docid}(s,r,P) = (T_{cpu}(s+ r+s)/P) * \text{LI}[P]$
Sort costs:	$\text{Sort\_Route}_{docid}(s,P) = (T_{cpu}(s/P)\mathit{log}(s/P)) * \text{LI}[P]$
Communication costs:	$\text{Comms\_Route}_{docid}(s,P,\text{SSIZE}) = (((s/\text{SSIZE})/P)+2P)* T_{comm}$

Putting these functions together gives us the evaluation cost model for *DocId* term selection:

$$\text{EVAL}_{docid}(s,r,i,q,P,\text{SSIZE}) = \text{INSPECTED}(q,i) * (\text{Merge\_Route}_{docid}(s,r,P) + \text{Sort\_Route}_{docid}(s,P) + \text{Comms\_Route}_{docid}(s,P,\text{SSIZE}))$$

We also measure overheads at the synchronisation point for merging the chosen term into the accumulated set and communicating the best term identifier in one iteration;

Communication costs for best term:	$P*T_{comm}$
Merge best term set into accumulated set:	$((s*T_{cpu})/P * \text{LI}[P])$

We assume latency is the dominant factor in communication costs. Putting these equations together gives us the estimate of overheads for the *DocId* term selection cost model.

$$\mathrm{OVERHEAD}_{docid}(s,i,P) = i*(((s*T_{cpu})/P) * LI[P]) + (P*T_{comm}))$$

The models for term selection are constructed by taking the load cost model (defined in section A5.7.3), and adding the evaluation and overhead cost models defined above in this section. The load cost model is further refined by dividing by the number of processors and factoring the result by the load imbalance estimate (LI[P]).

$$\mathrm{ROUTING}_{docid}(s,r,i,q,P,SSIZE) = \frac{\underline{LOAD(q,s)}}{P} * LI[P] + \mathrm{EVAL}_{docid}(s,r,i,q,P,SSIZE) + \mathrm{OVERHEAD}_{docid}(s,i,P)$$

$$\mathrm{ROUTING}_{docidw}(s,r,i,q,P,SSIZE,w) = \frac{\underline{LOAD(q,s)}}{P} * LI[P] + (w*\mathrm{EVAL}_{docid}(s,r,i,q,P,SSIZE)) + \mathrm{OVERHEAD}_{docid}(s,i,P)$$

A5.8.2 *TermId Models*

The interaction at the synchronisation point is more complicated than for the *DocId* models. This is because the data for the best term must be retrieved from the relevant node and merged into the accumulated set, which is then broadcast to all nodes. Overheads for *TermId* models are calculated as follows:

Get the identifier of best term in one iteration:	$P*T_{comm}$
Request for best term set:	$1* T_{comm}$
Retrieving best set from relevant node:	$s/SSIZE * T_{comm}$
Broadcast best set to all other nodes:	$(P-1 * s/SSIZE) * T_{comm}$
Merge the best term data into the accumulated set:	$sT_{cpu}+T_{i/o}[s]$

Put together, these equations form the cost model for routing overheads on the *TermId* partitioning scheme;

$$\mathrm{OVERHEAD}_{termid}(s,i,P,SSIZE) = i*((T_{comm}*((P+1)*(P*s/SSIZE)))+ (sT_{cpu}+T_{i/o}[s]))$$

Construction of the routing models can be done by re-using the basic term selection models and adding the $(\mathrm{OVERHEAD}_{termid})$cost;

$$\mathrm{ROUTING}_{termid}(s,r,i,q,P,SSIZE) = (\mathrm{ROUTING}_{par}(s,r,i,q,P) * LI[P]) + \mathrm{OVERHEAD}_{termid}(s,i,P,SSIZE)$$

$\text{ROUTING}_{termidw}(s,r,i,q,P,SSIZE,w) =$
$(\text{ROUTING}_{parw}(s,r,i,q,P,w) * LI[P]) + \text{OVERHEAD}_{termid}(s,i,P,SSIZE)$

The extra LI[P] here assumes that ROUTING_{termid} imbalance will probably be worse than ROUTING_{rep} in particular or other models in general. This is because terms are statically allocated to a node (see chapter 4, sub-section 4.4.2.3 for a discussion on term allocation schemes).

A5.8.3 *Replication Models*

Latency is presumed to be the main communication problem for the replication distribution scheme. The overheads for replication cost models are calculated as follows:

Get the identifier of best term from P processors:	$P* T_{comm}$
Send the identifier of best term to P processors:	$P* T_{comm}$
Merge the best term data into the accumulated set:	$(s*T_{cpu})+T_{i/o}[s]$

Putting these equations together gives us the following cost model;

$\text{OVERHEAD}_{rep}(s,i,P) = i*(sT_{cpu}+T_{i/o}[s] + (2P*T_{comm}))$

The cost models for the replication distribution scheme can be constructed by simply adding the overheads to the basic term selection cost models.

$\text{ROUTING}_{rep}(s,r,i,q,P) =$
$\text{ROUTING}_{par}(s,r,i,q,P) + \text{OVERHEAD}_{rep}(s,i,P)$

$\text{ROUTING}_{repw}(s,r,i,q,P,w) =$
$\text{ROUTING}_{parw}(s,r,i,q,P,w) + \text{OVERHEAD}_{rep}(s,i,P)$

A5.8.4 *On-the-fly distribution Models*

The overhead cost model for the On-the-fly distribution scheme is;

$\text{OVERHEAD}_{load}(q,s,SSIZE,P) = ((qs/SSIZE)+P) * T_{comm}$

There are also overhead costs at the synchronisation point for transferring set data, which is formed as follows;

Get the identifier of best term in one iteration:	$P*T_{comm}$
Broadcast best set to all nodes:	$(P * s/SSIZE) * T_{comm}$
Merge the best term data into the accumulated set:	$(s*T_{cpu})+T_{i/o}[s]$

Putting these equations together, we have the overhead at the synchronisation point;

$\text{OVERHEAD}_{large}(s,i,SSIZE,P) = i*((((P*(s/SSIZE))+P) * T_{comm})+(sT_{cpu}+T_{i/o}[s]))$

We cannot use the basic parallel term selection costs models, as some aspects of them (such as load) must be done sequentially. We apply a parallel cost model to the total evaluation cost, together with the load cost model defined in section A5.7.3 and the overhead cost models defined above in this section.

$$\text{ROUTING}_{\text{parfly}}(s,r,i,q,\text{SSIZE},P) = \text{LOAD}(q,s) + \text{OVERHEAD}_{\text{load}}(q,s,\text{SSIZE},P) + \text{OVERHEAD}_{\text{large}}(s,i,\text{SSIZE},P) + \frac{(\text{INSPECTED}(q,i) * \text{EVAL}(q,s,r)}{P} * \text{LI}[P])$$

$$\text{ROUTING}_{\text{parflyw}}(s,r,i,q,\text{SSIZE},P,w) = \text{LOAD}(q,s) + \text{OVERHEAD}_{\text{load}}(q,s,\text{SSIZE},P) + \text{OVERHEAD}_{\text{large}}(s,i,\text{SSIZE},P) + \frac{(\text{INSPECTED}(q,i) * \text{EVAL}(q,s,r) * w}{P} * \text{LI}[P])$$

A5.9 SEQUENTIAL MODELS FOR INDEX UPDATE

A5.9.1 *Adding a Document to the Buffer: Update Transaction*

The Client/Server Update model is formed by the following steps;

Scan Word and put in client Tree:	$n\,log(n) * T_{cpu}$
Marshalling/UnMarshalling term data:	$(n + n) * T_{cpu}$
Sending data:	$\lceil (n/\text{SSIZE}) \rceil * T_{comm}$
Merge word data with server buffer:	$n\,log(dict) * T_{cpu}$

Putting these equations together gives us a cost model for update on a single inverted file.

$$\text{UPDATE}_{seq}(n,dict,\text{SSIZE}) = T_{cpu}(n\,log(n) + 2n + n\,log(dict)) + T_{comm} * \lceil (n/\text{SSIZE}) \rceil$$

A5.9.2 *Transaction while index is updated*

The cost model for transaction is calculated by adding the contention factor c to the particular function being examined. The search cost model is taken from section A5.3 and the update cost model from the previous section.

$$\text{UPDATE}_{seqc}(n,dict,\text{SSIZE}) = (\text{UPDATE}_{seq}(n,dict,\text{SSIZE}) * c[1]) + \text{UPDATE}_{seq}(n,dict,\text{SSIZE})$$

$$\text{SEARCH}_{seqc}(s,q) = (\text{SEARCH}_{seq}(s,q) * c[1]) + \text{SEARCH}_{seq}(s,q)$$

A5.9.3 *Transaction estimate*

Taking the cost models defined in sections A5.9.1 and A5.9.2 we can construct a cost model for transactions where the percentage of total transaction time spent updating the index can be used. This allows us the vary the effect on transactions and study the theoretical performance penalty on transaction while doing a simultaneous index update. In the TRANSACTION_{seq} model we eliminate the contention by setting ro to zero, while ro set to 1 means that all transactions are affected by contention.

$$\text{TRANSACTION}_{seq}(ur,sr,ro,n,dict,s,q) = \frac{(1\text{-}ro(ur*\text{UPDATE}_{seq}(n,dict,SSIZE)+ sr*\text{SEARCH}_{seq}(s,q))) + (ro(ur*\text{UPDATE}_{seqc}(n,dict,SSIZE)+ sr*\text{SEARCH}_{seqc}(s,q)))}{ur + sr}$$

A5.9.4 *Reorganisation of Inverted File*

The reorganisation model is made up of the following synthetic cost functions;

Insert m buffer words in dict:	$\text{Insert\_Words\_buff}(m,b,dict) = T_{cpu}(m(log(dict/b)+b))$
Read t+m keyword lists from disk:	$\text{List\_Disk\_Trans}(t,m) = T_{i/o}[s] * (t+m)$
Write t+m keyword lists to disk:	$\text{List\_Disk\_Trans}(t,m)$
Merge m keyword lists:	$\text{Merge\_kw\_lists}(m,s) = T_{cpu}(m(s+s))$
Read in (dict/b) keyword blocks:	$\text{Read\_kw\_blocks}(dict,b) = T_{i/o}[b] * (dict/b)$

The reorganisation or index update cost model is constructed by using the four cost models defined above (List_Disk_Trans is used twice);

$$\text{REORG}_{seq}(n,dict,b,m,t,s) = \text{Insert\_Words\_buff}(m,b,dict) + 2\text{List\_Disk\_Trans}(t,m) + \text{Merge\_kw\_lists}(m,s) + \text{Read\_kw\_blocks}(dict,b)$$

The contention model for reorganising the index is;

$$\text{REORG}_{seqc}(n,dict,b,m,t,s) = (\text{REORG}_{seq}(n,dict,b, m,t,s) * c[1]) + \text{REORG}_{seq}(n,dict,b,m,t,s)$$

A5.10 PARALLEL MODELS FOR INDEX UPDATE

A5.10.1 *DocId Transaction Model*

In this data distribution method we simply re-use the sequential cost model defined in section A5.9.1 above;

$$\text{UPDATE}_{docid}(n,dict,SSIZE) = \text{UPDATE}_{seq}(n,dict,SSIZE)$$

The contention model also re-uses the sequential model;

$$\text{UPDATE}_{docidc}(n,dict,SSIZE,P) = (\text{UPDATE}_{seq}(n,dict,SSIZE) * c[P]) + \text{UPDATE}_{seq}(n,dict,SSIZE)$$

In order to construct the contention model for *DocId* search we re-use the function defined in section A5.4.1 above;

$$\text{SEARCH}_{docidc}(s,q,P) = (\text{SEARCH}_{docid}(s,q,P) * c[P]) + \text{SEARCH}_{docid}(s,q,P)$$

The transaction model for *DocId* partitioning is constructed in exactly the same way as the sequential version described in section A5.9.3 above.

$$\text{TRANSACTION}_{docid}(ur,sr,ro,n,dict,s,q,P) =$$

$$\frac{(1\text{-ro}(\text{ur}*\text{UPDATE}_{docid}(\text{n,dict,P})+\text{ sr}*\text{SEARCH}_{docid}(\text{s,q,P}))) + (\text{ro}(\text{ur}*\text{UPDATE}_{docidc}(\text{n,dict,SSIZE,P})+\text{ sr}*\text{SEARCH}_{docidc}(\text{s,q,P})))}{\text{ur + sr}}$$

A5.10.2 *TermId Transaction Model*

With the TermId distribution method, a new cost model must be defined as merging the data with the buffer is parallelised.

$$\text{UPDATE}_{termid}(\text{n,dict,P,SSIZE}) = T_{cpu}(n\,log(n) + \frac{n\,log(dict)}{P} *LI[P]) + 2n\,) + (P*T_{comm}*\lceil(n/SSIZE)\rceil)$$

The contention model re-uses the model defined above;

$$\text{UPDATE}_{termidc}(\text{n,dict,P,SSIZE}) = (\text{UPDATE}_{termid}(\text{n,dict,P,SSIZE}) * c[P]) + \text{UPDATE}_{termid}(\text{n,dict,P,SSIZE})$$

In order to construct the contention model for *TermId* search we re-use the function defined in section A5.4.3 above (we utilise the parallel sort cost model);

$$\text{SEARCH}_{termidc}(\text{s,q,P,SSIZE}) = (\text{SEARCH}_{termid2}(\text{s,q,P,SSIZE}) * c[P]) + \text{SEARCH}_{termid2}(\text{s,q,P,SSIZE})$$

The transaction model for *DocId* partitioning is constructed in exactly the same way as the sequential version described in section A5.9.3 above.

$$\text{TRANSACTION}_{termid}(\text{ur,sr,ro,n,dict,s,q,P,SSIZE}) = \frac{(1\text{-ro}(\text{ur}*\text{UPDATE}_{termid}(\text{n,dict,P})+\text{ sr}*\text{SEARCH}_{termid2}(\text{s,q,P,SSIZE}))) + (\text{ro}(\text{ur}*\text{UPDATE}_{termidc}(\text{n,dict,P,SSIZE})+\text{ sr}*\text{SEARCH}_{termidc}(\text{s,q,P,SSIZE})))}{\text{ur + sr}}$$

A5.10.3 *DocId Reorganisation Model*

The *DocId* index update cost model is;

Insert m buffer words in dict:	$\text{Insert\_Words\_buff}_{docid}(\text{m,b,dict,P}) = T_{cpu}(m*i[P]*\,(log(((dict/b)/P)*i[P])+b)) * LI[P]$
Read t+m keyword lists from disk:	$\text{List\_Disk\_Trans}_{docid}(\text{t,m,s,P}) = T_{i/o}[p[P]*(s/P)] * (t+m) *i[P] * LI[P]$
Write t+m keyword lists to disk:	$\text{List\_Disk\_Trans}_{docid}(\text{t,m,s,P})$
Merge m keyword lists:	$\text{Merge\_kw\_lists}_{docid}(\text{m,s,P}) = T_{cpu}(m*i[P]*(s/P +s/P)) * LI[P]$
Read in (dict/b) keyword blocks:	$\text{Read\_kw\_blocks}_{docid}(\text{dict,b,P}) = T_{i/o}[b] * ((dict/b)/P) *i[P] * LI[P]$

The index update model for *DocId* partitioning is constructed as follows;

$$\text{REORG}_{docid}(\text{n,dict,b,m,t,s,P}) = \text{Insert\_Words\_buff}_{docid}(\text{m,b,dict,P}) + 2\text{List\_Disk\_Trans}_{docid}(\text{t,m,s,P}) + \text{Merge\_kw\_lists}_{docid}(\text{m,s,P}) + \text{Read\_kw\_blocks}_{docid}(\text{dict,b,P})$$

This function is re-used in the construction of the cost model with contention as follows;

$$\text{REORG}_{docidc}(\text{n,dict,b,m,t,s,P}) = (\text{REORG}_{docid}(\text{n,dict,b,m,t,s,P}) *c[P]) + \text{REORG}_{docid}(\text{n,dict,b,m,t,s,P})$$

A5.10.4 *TermId Reorganisation Model*

The *TermId* index update cost model is;

Insert m buffer words in dict:	$\text{Insert\_Words\_buff}_{termid}(m,b,dict,P) = (T_{cpu}(m(log(dict/b)+b))/P) * LI[P]$
Read t+m keyword lists from disk:	$\text{List\_Disk\_Trans}_{termid}(t,m,s,P) = ((T_{i/o}[s]* (t+m))/P) * LI[P]$
Write t+m keyword lists to disk:	$\text{List\_Disk\_Trans}_{termid}(t,m,s,P)$
Merge m keyword lists:	$\text{Merge\_kw\_lists}_{termid}(m,s,P) = (T_{cpu}(m(s+s))/P) * LI[P]$
Read in (dict/b) keyword blocks:	$\text{Read\_kw\_blocks}_{termid}(dict,b,P) = ((T_{i/o}[b] * (dict/b))/P) * LI[P]$

The index update model for *TermId* partitioning is constructed as follows;

$$\text{REORG}_{termid}(n,dict,b,m,t,s,P) = \text{Insert\_Words\_buff}_{termid}(m,b,dict,P) + 2\text{List\_Disk\_Trans}_{termid}(t,m,s,P) + \text{Merge\_kw\_lists}_{termid}(m,s,P) + \text{Read\_kw\_blocks}_{termid}(dict,b,P)$$

This function is re-used in the construction of the cost model with contention as follows;

$$\text{REORG}_{termidc}(n,dict,b,m,t,s,P) = (\text{REORG}_{termid}(n,dict,b,m,t,s,P) *c[P]) + \text{REORG}_{termid}(n,dict,b,m,t,s,P)$$