Parallel methods for the update of partitioned inverted files

A. MacFarlane
School of Informatics, City University London, UK

J.A. McCann
Department of Computing, Imperial College London, UK

S.E. Robertson
Microsoft Research Ltd, Cambridge, UK

Abstract
Purpose – An issue which tends to be ignored in information retrieval is the issue of updating inverted files. This is largely because inverted files were devised to provide fast query service, and much work has been done with the emphasis strongly on queries. In this paper we study the effect of using parallel methods for the update of inverted files in order to reduce costs, by looking at two types of partitioning for inverted files: document identifier and term identifier.
Design/methodology/approach – Raw update service and update with query service are studied with these partitioning schemes using an incremental update strategy. We use standard measures used in parallel computing such as speedup to examine the computing results and also the costs of reorganising indexes while servicing transactions.
Findings – Empirical results show that for both transaction processing and index reorganisation the document identifier method is superior. However, there is evidence that the term identifier partitioning method could be useful in a concurrent transaction processing context.
Practical implications – There is an increasing need to service updates which is now becoming a requirement of inverted files (for dynamic collections such as the Web), demonstrating that a shift in requirements of inverted file maintenance is needed from the past.
Originality/value – The paper is of value to database administrators who manage large-scale and dynamic text collections, and who need to use parallel computing to implement their text retrieval services.
Keywords Information retrieval, Parallel programming, Inverted files
Paper type Research paper

1. Introduction
One of the most neglected areas in information retrieval is the issue of servicing updates to inverted files. In most applications this is understandable given that some databases will not be updated very frequently: for example, Dialog and DataStar have databases which are updated weekly, monthly quarterly or even yearly (Thomson Dialog, 2005). However, some applications such as web search engines or News services like Reuters could have updates arriving 24 hours a day, and there is no time when the system could be taken down and the updates serviced in a batch. 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 services on inverted files. In this paper we describe the update mechanism for a Parallel text retrieval system, PLIERS, on two different types of partitioning methods: term identifier partitioning (*TermId)* where each term, with all its associated postings is assigned to a single fragment (and therefore information about any document is distributed among fragments); and document identifier partitioning (*DocId*) where the reverse is the case.

2. Experimental methodology

Much of the previous work in the area of inverted file maintenance (Reddaway, 1991; Shoens *et al*., 1994; Clark and Cormack, 1995; Brown *et al*., 1994) has advocated the use of buffering updates to save on Input/Output. 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 update index with the index kept on disk. In order to do this we use the following strategy:

1. Read in inverted list from disk.
2. Add new postings to inverted list.
3. 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: we do not address the issue of document removals or document changes. Our assumption is that text collections are in the main archival. Our priority is to try and keep the index in a state that would allow us to service fast query processing. 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. There may therefore be some delays to transactions while a reorganisation of the index is going on.

A number of issues have not been addressed, such as simultaneous update and query processing together with concurrency control due to time constraints. However, we recognise the importance of those issues and deal with them theoretically elsewhere (MacFarlane *et al*., 1996). The document availability semantics we use is *Late Availability* which is defined by MacFarlane and colleagues (1996). A survey of strategies for updating inverted files is available in Zobel and Moffat (2006).

3. Transaction topologies

Given that we want to service search and updates simultaneously, the transaction topology cannot differ too much from the search topology described elsewhere (MacFarlane *et al*., 2000). We therefore define top and leaf nodes which can handle both search operations and the update operations implemented. We describe the additional functionality needed by the nodes to support update operations above. Figure 1 shows an example of transaction topology with the service of updates.

Take in Figure 1. Example of transaction topology configuration

3.1 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. The main issue here is that the top node must know what type of partitioning method 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 make the assumption 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 document to fragments using a "round robin" distribution method when document identifier 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.

3.2 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 another paper by MacFarlane and colleagues (2005). 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 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. When the reorganisation is complete this temporary buffer becomes the main buffer.

4. Software and hardware used

PLIERS (ParaLLel Informaton rEtrieval Rearch System) has been developed at City University using ideas from Okapi to investigate the use of parallelism in IR. PLIERS is designed to run on several parallel architectures and is currently implemented on those which use Sun Sparc, DEC Alpha and Pentium PII processors. The results presented in this paper were obtained on 8 nodes of a 12 node AP3000 at the Australian National University, Canberra. Each node has its own local disk: the

shared nothing architecture (DeWitt and Gray, 1992) is used by PLIERS. The Fujistsu AP3000 is a distributed memory parallel Computer using Ultra 1 processors running Solaris 2.5.1. Each node has a speed of 167Mhz. The torus network has a top bandwidth of 200 Mbytes/s per second.

5. Data and settings used

The data used in the experiments was the BASE1 and BASE10 collections, both sub-sets of the official 100 Gigabyte VLC2 collection (Hawking *et al*., 1999). The BASE1 is 1 Gigabyte in size, while BASE10 is approximately 10 Gigabytes in size. We use two types of builds for indexes: *distributed builds* where text is kept centrally and distributed to index nodes and *local builds* where text is physically distributed to nodes and indexed locally (MacFarlane *et al*., 2005). For the *distributed build* method we use the BASE1 collection only, creating indexes on 1 to 7 processors and servicing transactions on all of those indexes. Two types of index were 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. The *DocId* partition method is used on these experiments. We built one set of indexes which contained position data and one set without position data for both types of build methods and both types of partitioning methods (runs with position data are marked ‘position data’, while those without such data are marked ‘postings only’).

Take in Table I. Details of transaction sets used in experiments

Table I shows the transaction sets used in our experiments. 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 only 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-22173 collection (Lewis, 2006) not in the VLC2 set: we refer to this file as REUTERS. We chose this set because we can guarantee that the data is new to the VLC2 set. The REUTERS file is 1.2 Mb in size and has 1000 records. We took both these sets and created transaction sets with differing numbers of updates and queries, varying the number of updates to queries. We do this at a number of rates ranging from 10 queries per 1 update down to 1 query per 1.25 updates. We also examine update and query only service. This allows us to both examine the effect between updates and queries as well as finding a good point where buffer re-organisation is needed. We apply these transactions to all the indexes built (described above), both in the presence and absence of an index reorganisation. All figures produced are averages of 5 runs per experiment. For the one leaf experiments we use a client/server process. We record the raw index reorganisation speed to establish the best point to initiate it. We use a number of measures to examine the results. These are elapsed time in seconds, load imbalance (LI), speedup, and scaleability for transactions and index reorganisation and transaction throughput (transactions per hour). Equations for most of these metrics are declared in the Glossary.

6. 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 6.1). Is there a big performance penalty in only allowing one update at a time in the system? 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 6.2). Both updates and transactions are examined in the presence and absence of index reorganisation. The performance of index reorganisation is examined in section 6.3, together with a discussion on a good buffer size for the collections being examined. A summary of the experimental results is given in section 6.4.

Take in Figure 2. BASE1 [*DocId*]: average elapsed time in ms for update transactions (postings only)
Take in Figure 3. BASE1 [*TermId*]: average elapsed time in ms for update transactions (postings only)
Take in Figure 4. BASE1 [*DocId*]: average elapsed time in ms for update transactions (position data)
Take in Figure 5. BASE1 [*TermId*]: average elapsed time in ms for update transactions (position data

6.1 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 Figures 2 to 5). 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 transactions 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 Figures 6 to 9).

Take in Figure 6. BASE1 [*DocId*]: speedup for update transactions (postings only)
Take in Figure 7. BASE1 [*TermId*]: speedup for update transactions (postings only)
Take in Figure 8. BASE1 [*DocId*]: speedup for update transactions (position data)
Take in Figure 9. BASE1 [*TermId*]: speedup for update transactions (position data)

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 Figure 5).

Take in Figure 10. BASE1 [*DocId*]: average elapsed time in ms for update transactions during index reorganisation (postings only)
Take in Figure 11. BASE1 [*TermId*]: average elapsed time in ms for update transactions during index reorganisation (postings only)
Take in Figure 12. BASE1 [*DocId*]: average elapsed time in ms for update transactions during index reorganisation (position data)
Take in Figure 13. BASE1 [*TermId*]: average elapsed time in ms for update transactions during index reorganisation (position data)

Figures 10 to 13 show 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 Figures 14 to 17. 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 three hundred per cent over elapsed times when index reorganisation is done.

Take in Figure 14. BASE1 [*DocId*]: % increase in average elapsed time for update transactions during index reorganisation (postings only)
Take in Figure 15. BASE 1 [*Termld*]: % increase in average elapsed time for update transactions during index reorganisation (postings only)
Take in Figure 16. BASE 1 [*Docld*] % increase in average elapsed time for update transactions during index reorganisation (position data)
Take in Figure 17. BASE 1 [*Termld*]: % increase in average elapsed time for update transactions during index reorganisation (position data)

Take in Table II. BASE1/BASE10 [*DocId*]: index update results for update transactions

Table II 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.

6.2 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. Figures 18 to 21 show average elapsed times for transactions on the BASE1 collection using all types of indexes and partitioning methods.

Take in Figure 18. BASE1 [*DocId*]: transaction average elapsed times in ms (postings only)
Take in Figure 19. BASE1 [*TermId*]: transaction average elapsed times in ms (postings only)
Take in Figure 20. BASE1 [*DocId*]: transaction average elapsed times in ms (position data)
Take in Figure 21. 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 (MacFarlane *et al*., 2000) severely affect 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 Figure 23). Speed advantage on indexes containing position data is recorded, but is very slight (see Figure 25). With *DocId* partitioning we do gain speed advantage using parallelism (see Figures 22 and 24), but the proportion of updates in the transaction set may actually increase the average elapsed time when more leaf nodes are used (see Figures 18 and 20). 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.

Take in Figure 22. BASE 1 [*Docld*]: speedup for all transactions (postings only)

Take in Figure 23. BASE 1 [*Termld*]: speedup for all transactions (postings only)
Take in Figure 24. BASE 1 [*Docld*]: speedup for all transactions (position data)
Take in Figure 25. BASE 1 [*Termld*]: speedup for all transactions (position data)

Figures 26 to 29 show the effect of index reorganisation on transactions serviced over BASE1 collection. 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 Figures 27 and 29). 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.

Take in Figure 26. BASE1 [*DocId*]: average elapsed time in ms for all transactions during index reorganisation (postings only)
Take in Figure 27. BASE1 [*TermId*]: average elapsed time in ms for all transactions during index reorganisation (postings only)
Take in Figure 28. BASE1 [*DocId*]: average elapsed time in ms for all transactions during index reorganisation (position data)
Take in Figure 29. BASE1 [*TermId*]: average elapsed time in ms for all transactions during index reorganisation (position data)

What effect do these results have on throughput? In Figures 30 to 34 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.

Take in Figure 30. BASE1: combined transactions throughput for UPDATE1 transaction set
Take in Figure 31. BASE1: combined transactions throughput for UPDATE2 transaction set
Take in Figure 32. BASE1: combined transactions throughput for UPDATE3 transaction set
Take in Figure 33. BASE1: combined transactions throughput for UPDATE4 transaction set
Take in Figure 34. 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 Figures 30 and 34). Throughput on the index type/partitioning method relative to each other is consistent irrespective of the transaction set under scrutiny.

Take in Figure 35. BASE1 [*DocId*]: load imbalance for all transactions (postings only)
Take in Figure 36. BASE1 [*DocId*]: load imbalance for all transactions (position data)
Take in Figure 37. BASE1 [*TermId*]: load imbalance for all transactions (postings only)
Take in Figure 38. 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: Figures 35 to 38 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 (MacFarlane *et al.*, 2000)).

Take in Table III. BASE1/BASE10 [*DocId*]: index update results for all transactions

Table III 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 II). Elapsed times trends are inverse to that of scalability and for the same reason.

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 paper 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 (MacFarlane *et al.*, 2000). 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.

6.3 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). Figures 39 to 42 show the index reorganisation results using elapsed time in seconds.

Take in Figure 39. BASE1 [*DocId*]: index reorganisation elapsed time in seconds (postings only)
Take in Figure 40. BASE1 [*DocId*]: index reorganisation elapsed time in seconds (position data)
Take in Figure 41. BASE1 [*TermId*]: index reorganisation elapsed time in seconds (postings only)
Take in Figure 42. 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 documents 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. Figures 43 to 46 show the speedup for index reorganisation on both partitioning methods.

Take in Figure 43. BASE1 [*DocId*]: index reorganisation speedup (postings only)
Take in Figure 44. . BASE1 [*DocId*]: index reorganisation speedup (position data)
Take in Figure 45. BASE1 [*TermId*]: index reorganisation speedup (postings only)
Take in Figure 46. 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 an 80 document buffer is particularly disappointing considering the other results. We will return to this factor when discussing load imbalance below. Why does this super-linear speedup 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. Figures 47 and 48 show the underlying reason why the super-linear speedup occurs.

Take in Figure 47. BASE1 [*DocId*]: millions of postings handled during index reorganisation
Take in Figure 48. BASE1 [*TermId*]: millions of postings handled during index reorganisation

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.

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 49 to 52 provides more evidence of the increased buffer effect on load imbalance.

Take in Figure 49. BASE1 [*DocId*]: index reorganisation load imbalance (postings only)
Take in Figure 50. BASE1 [*TermId*]: index reorganisation load imbalance (postings only)
Take in Figure 51. BASE1 [*DocId*]: index reorganisation load imbalance (position data
Take in Figure 52. BASE1 [*TermId*]: index reorganisation load imbalance (position data)

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. This has a direct and significant impact on speedup for *TermId* partitioning runs for 6 leaf nodes (see figs 45 and 46).

6.4 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 previous experiments with probabilistic search (MacFarlane *et al*., 2000). 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 paper. 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 paper. This issue will be examined further in the conclusion.

Conclusion

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. Problems highlighted in our probabilistic search experiments (MacFarlane *et al*., 2000) 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. 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 paper 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 an 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.*, 1990) which allow for cheap updates, but also allow for a high degree of parallelism.

Acknowledgements

This research is supported by the British Academy under grant number IS96/4203. We are also grateful to ACSys for awarding the first author a visiting student fellowship at the Australian National University in order to complete this research and the use of their equipment. We are particularly grateful to David Hawking for making the arrangements for the visit to the ANU.

References

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. (Eds), *Proceedings of EDBT'94*, March 1994, LNCS 779, Springer-Verlag, pp. 365-377.

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, University of Waterloo, Ontario.

Cringean, J.K., England, R., Manson, G.A. and Willett, P. (1990), "Parallel text searching in serial files using a processor farm", in Vidick, J.L. (Ed), *Proceedings of the 13th International Conference on Research and Development in Information Retrieval*, ACM Press, pp. 429-453.

DeWitt D. and Gray, J. (1992), "Parallel database systems: the future of high performance database systems", *Communications of the ACM*, Vol. 35, No. 6.

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*, Prentice-Hall, pp. 1-12.

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 the Seventh Text Retrieval Conference, Gaithersburg, U.S.A, November 1998, NIST Special publication 500-242, pp. 91-104.

Lewis, D. (2006), The Reuters 21578 test collection, available at: http://www.daviddlewis.com/resources/testcollections/reuters21578/ (accessed 22 August 2006).

MacFarlane, A., McCann J.A. and Robertson S.E. (2005), "Parallel methods for the generation of partitioned inverted files", *Aslib Proceedings*, Vol. 57 No. 5, pp. 434-459.

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*, BCS IRSG, pp. 67-79.

MacFarlane, A., Robertson S.E. and McCann J.A. (1999), "PLIERS at TREC8", in Voorhees, E. and Harman, D.K. (Eds), *Proceedings of the Eight Text Retrieval Conference, Gaithersburg, U.S.A, November 1999*, Gaithersburg, SP 500-246, NIST, Gaithersburg, pp. 241-252.

MacFarlane, A., Robertson S.E. and McCann J.A. (2000), "Parallel methods for the search of partitioned inverted files, in, De La Fuente, P. (Ed.), *Proceedings of String Processing and Information Retrieval - SPIRE 2000*, September 2000, A Coruna, Spain, IEEE Computer Society Press, pp. 209-220.

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*, Springer-Verlag, London, pp. 329-338.

Reddaway, S.F. (1991), "High speed text retrieval from large databases on a massively parallel processor", *Information Processing & Management*, Vol. 27 No. 4, pp. 311-316.

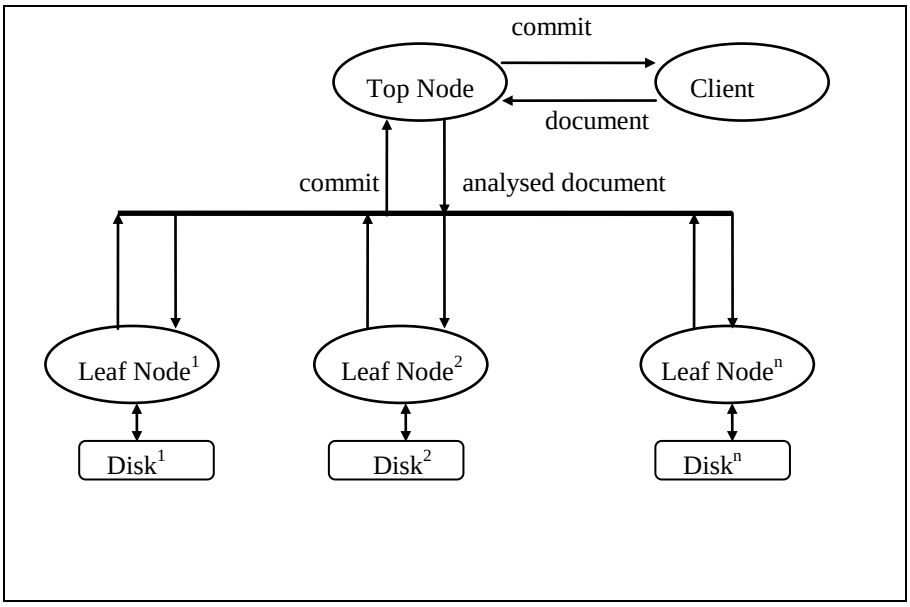
Thomson Dialog (2005), Dialog Database Catalog, Thomson Dialog Ltd, http://www.dialog.com [visited 29th May 2007]

Zobel, J. and Moffat, A. (2006), "Inverted files for text search engines", *ACM Computing Surveys*, Vol. 38, No. 2, article 6.

Glossary

Distributed Build	Method of building indexes where text is distributed from a single node.
DocId	Parititioning method which assigns all document data for a given document to one index partition
Efficiency	Measure of the effective use of processors. Definition: Speedup on n processors/n processors
LI	A measure of the amount of load imbalance on n processors: max time on n processors/average time on n processors
Local Build	Method of indexing where all processing is kept local to the node.
Mhz	Megahertz: processor clock speed.
Partition	Fragment of inverted file on a nodes disk.
Scalability	A measure of how well the algorithm scales on the same equipment. Definition: $\frac{\text{Time on small collection}}{\text{Time on large collection}} * \frac{\text{Size of large collection}}{\text{Size of small collection}}$
Speedup	Measure of speed advantage of parallelism. Definition: Time on 1 processors / Time on n processors.
TermId	Parititioning method which assigns all term data for a given term to one partition
TREC	Annual Text Retrieval Conference run by the National Institute of Standards and Technology in the United States.
VLC	Very Large Collection: Collection of 100 GB web data used in the TREC-7 VLC2 sub-track.

Figure 1. Example of transaction topology configuration



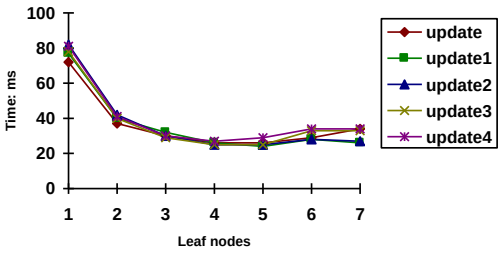


Figure 2. BASE1 [*DocId*]: average elapsed time in ms for update transactions (postings only)

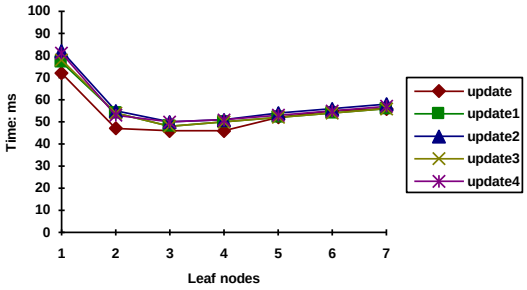


Figure 3. BASE1 [*TermId*]: average elapsed time in ms for update transactions (postings only)

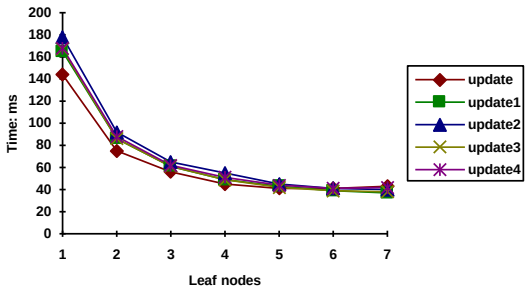


Figure 4. BASE1 [*DocId*]: average elapsed time in ms for update transactions (position data)

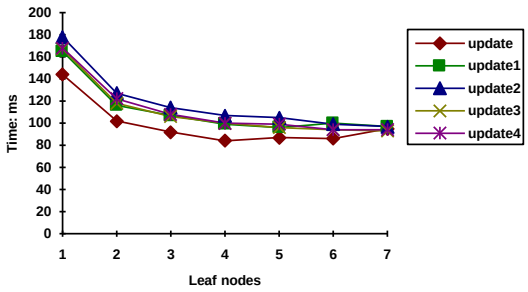


Figure 5. BASE1 [*TermId*]: average elapsed time in ms for update transactions (position data)

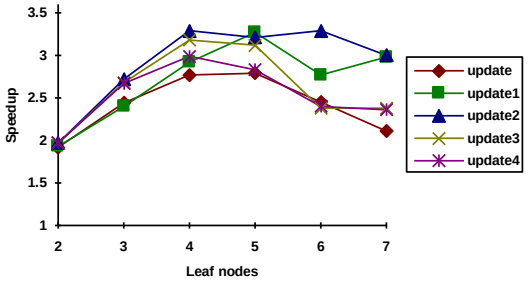


Figure 6. BASE1 [*DocId*]: speedup for update transactions (postings only)

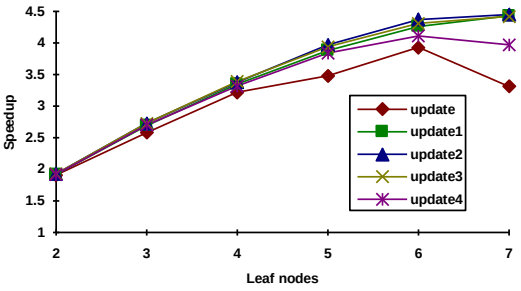


Figure 8. BASE1 [*DocId*]: speedup for update transactions (position data)

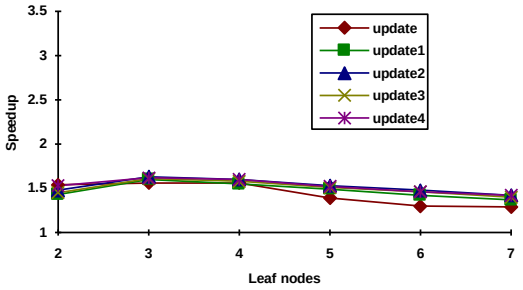


Figure 7. BASE1 [*TermId*]: speedup for update transactions (postings only)

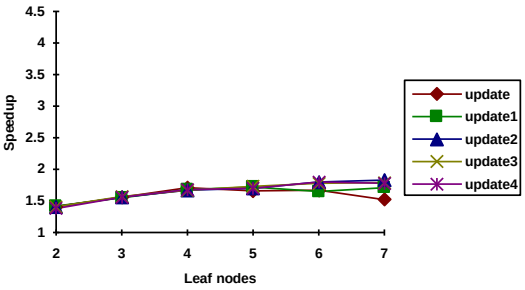


Figure 9. BASE1 [*TermId*]: speedup for update transactions (position data)

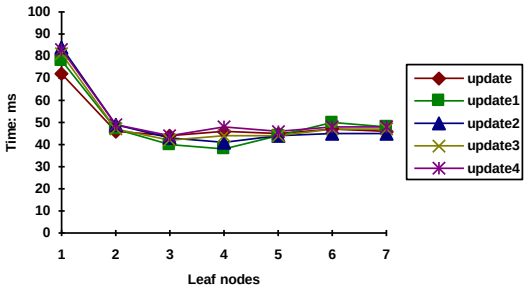


Figure 10. BASE1 [*DocId*]: average elapsed time in ms for update transactions during index reorganisation (postings only)

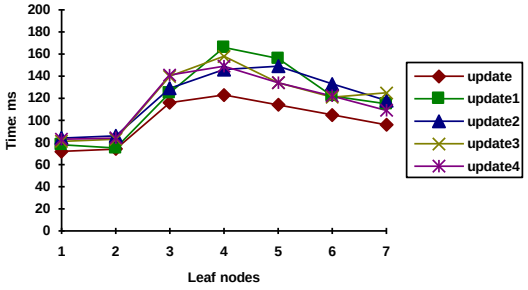


Figure 11. BASE1 [*TermId*]: average elapsed time in ms for update transactions during index reorganisation (postings only)

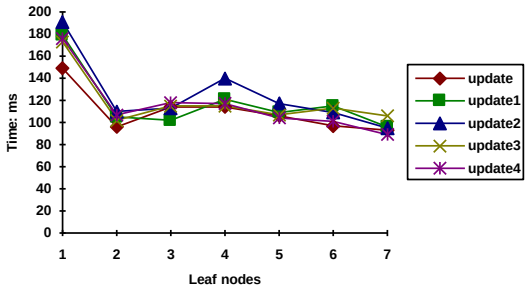


Figure 12. BASE1 [*DocId*]: average elapsed time in ms for update transactions during index reorganisation (position data)

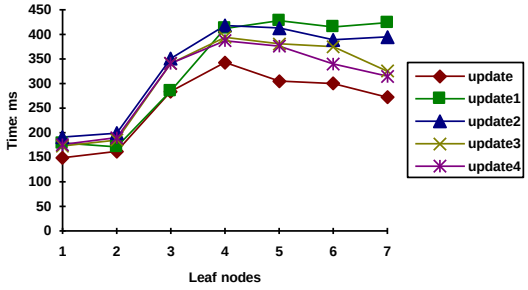


Figure 13. BASE1 [*TermId*]: average elapsed time in ms for update transactions during index reorganisation (position data)

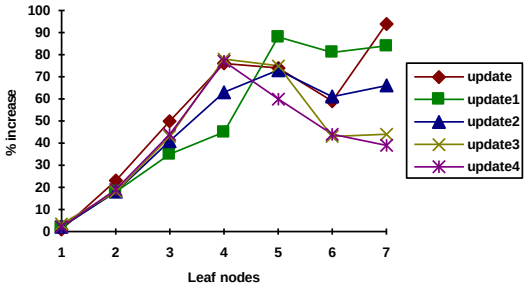


Figure 14. BASE1 [*DocId*]: % increase in average elapsed time for update transactions during index reorganisation (postings only)

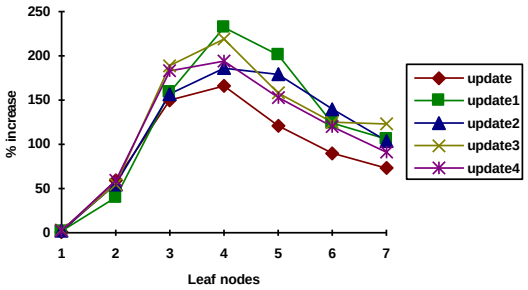


Figure 15. BASE1 [*TermId*]: % increase in average elapsed time for update transactions during index reorganisation (postings only)

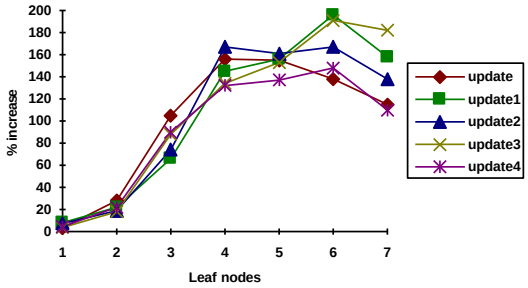


Figure 16. BASE1 [*DocId*]: % increase in average elapsed time for update transactions during index reorganisation (position data)

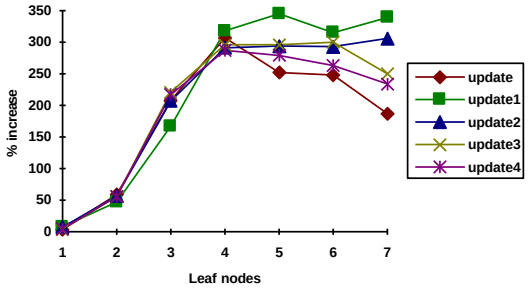


Figure 17. BASE1 [*TermId*]: % increase in average elapsed time for update transactions during index reorganisation (position data)

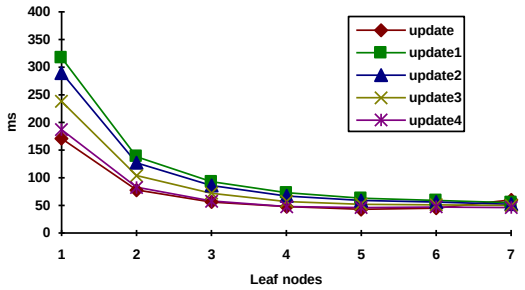


Figure 18. BASE1 [*DocId*]: transaction average elapsed times in ms (postings only)

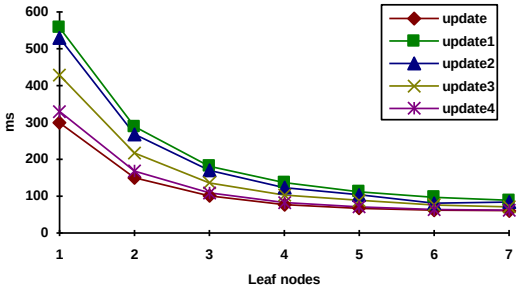


Figure 20. BASE1 [*DocId*]: transaction average elapsed times in ms (position data)

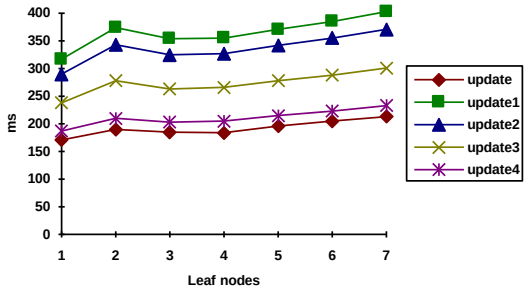


Figure 19. BASE1 [*TermId*]: transaction average elapsed times in ms (postings only)

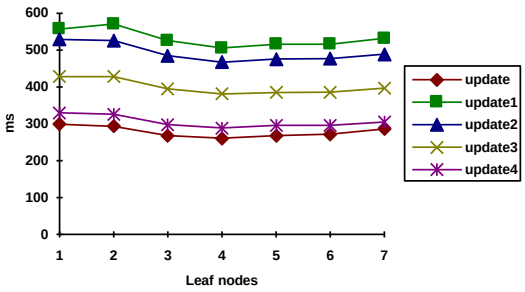


Figure 21. BASE1 [*TermId*]: transaction average elapsed times in ms (position data)

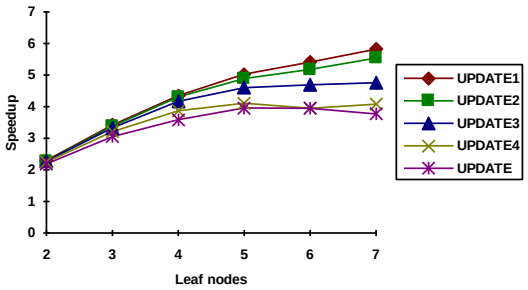


Figure 22. BASE1 [*DocId*]: speedup for all transactions (postings only)

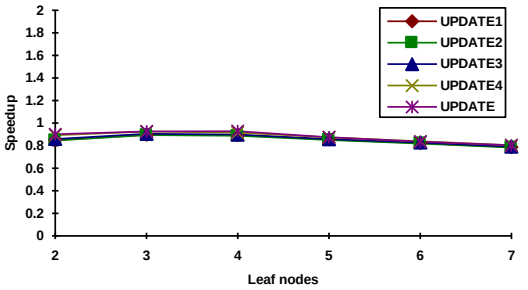


Figure 23. BASE1 [*TermId*]: speedup for all transactions (postings only)

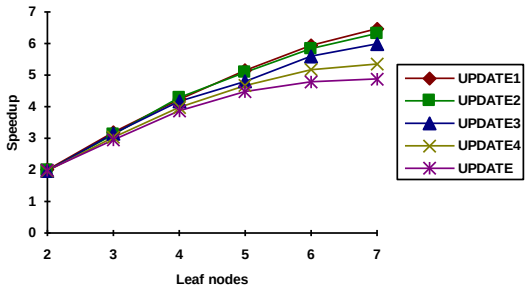


Figure 24. BASE1 [*DocId*]: speedup for all transactions (position data)

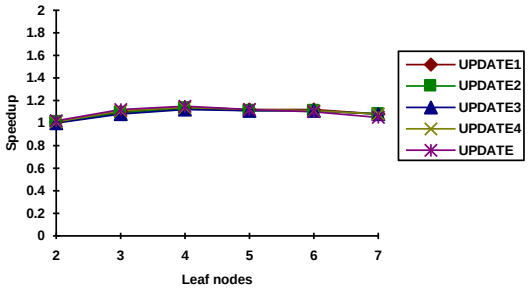


Figure 25. BASE1 [*TermId*]: speedup for all transactions (position data)

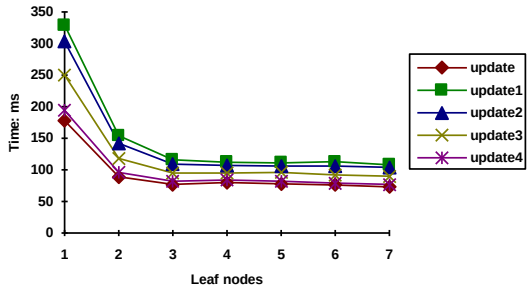


Figure 26. BASE1 [*DocId*]: average elapsed time in ms for all transactions during index reorganisation (postings only)

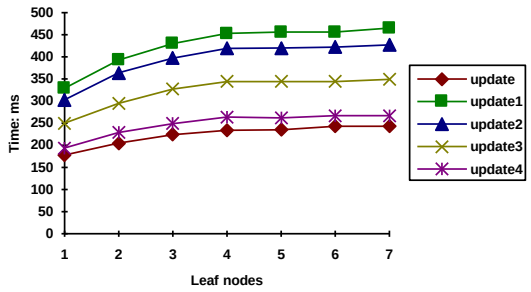


Figure 27. BASE1 [*TermId*]: average elapsed time in ms for all transactions during index reorganisation (postings only)

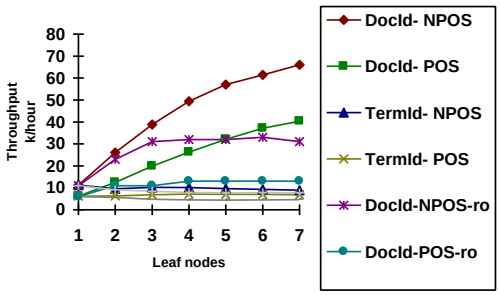


Figure 30. BASE1: combined transactions throughput for UPDATE1 transaction set

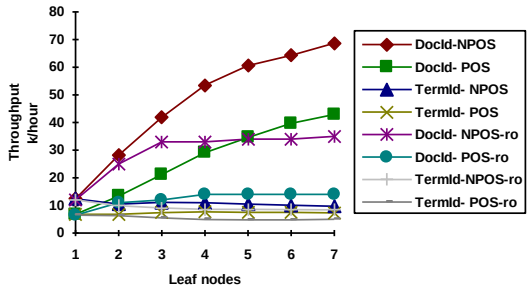


Figure 31. BASE1: combined transactions throughput for UPDATE2 transaction set

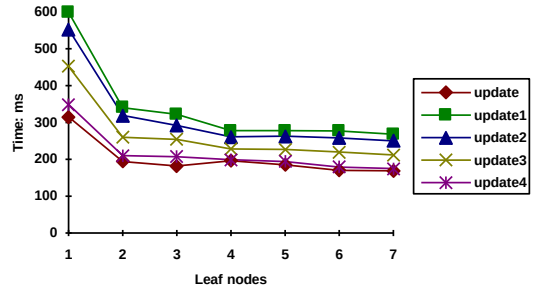


Figure 28. BASE1 [*DocId*]: average elapsed time in ms for all transactions during index reorganisation (position data)

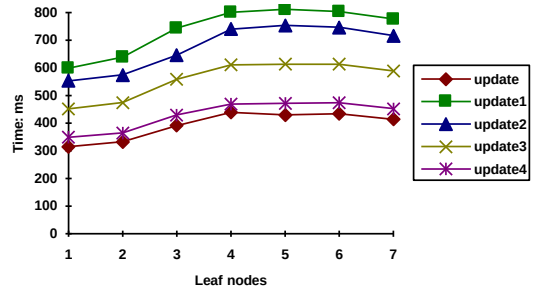


Figure 29. BASE1 [*TermId*]: average elapsed time in ms for all transactions during index reorganisation (position data)

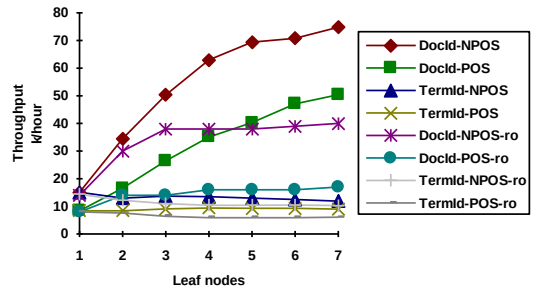


Figure 32. BASE1: combined transactions throughput for UPDATE3 transaction set

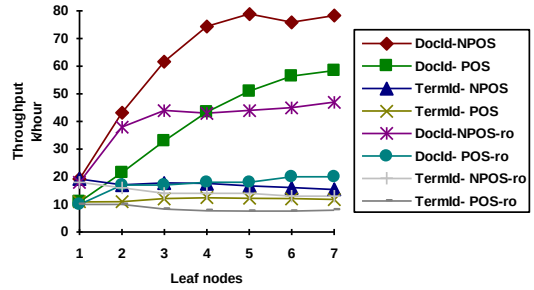


Figure 33. BASE1: combined transactions throughput for UPDATE4 transaction set

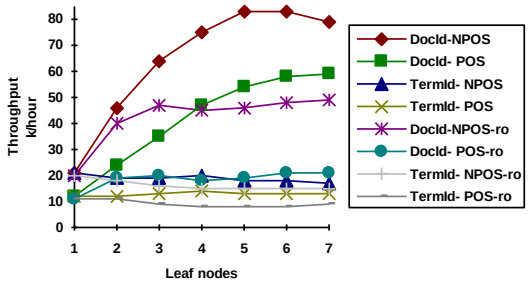


Figure 34. BASE1: combined transactions throughput for UPDATE transaction set

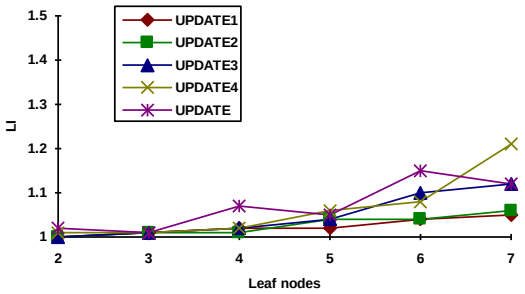


Figure 35. BASE1 [*DocId*]: load imbalance for all transactions (postings only)

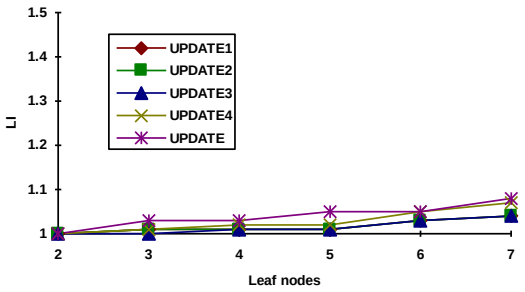


Figure 36. BASE1 [*DocId*]: load imbalance for all transactions (position data)

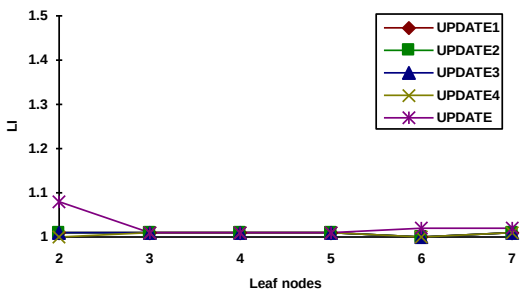


Figure 37. BASE1 [*TermId*]: load imbalance for all transactions (postings only)

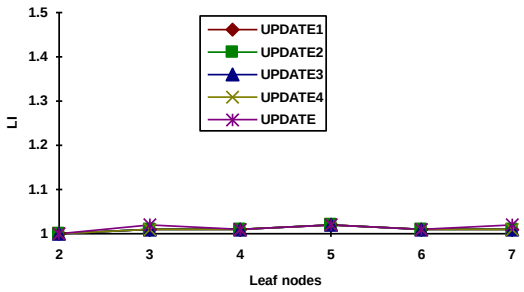


Figure 38. BASE1 [*TermId*]: load imbalance for all transactions (position data)

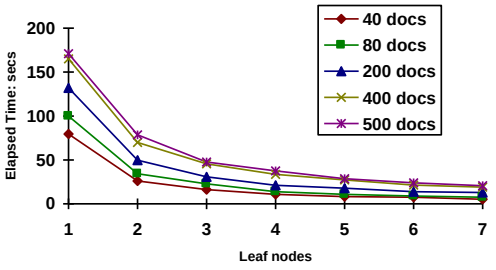


Figure 39. BASE1 [*DocId*]: index reorganisation elapsed time in seconds (postings only)

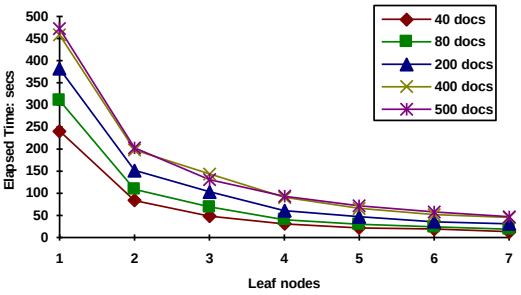


Figure 40. BASE1 [*DocId*]: index reorganisation elapsed time in seconds (position data)

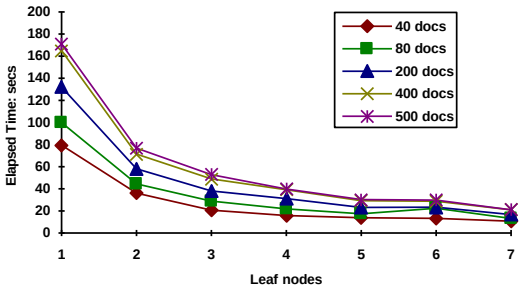


Figure 41. BASE1 [*TermId*]: index reorganisation elapsed time in seconds (postings only)

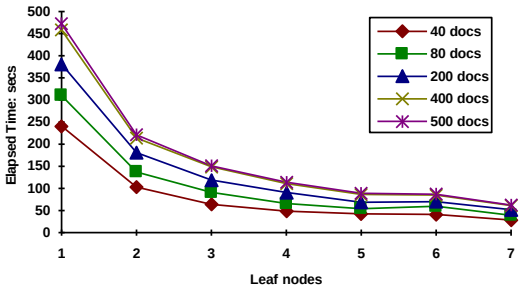


Figure 42. BASE1 [*TermId*]: index reorganisation elapsed time in seconds (position data)

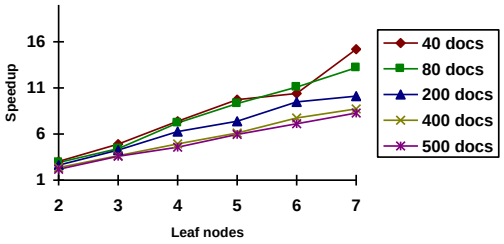


Figure 43. BASE1 [*DocId*]: index reorganisation speedup (postings only)

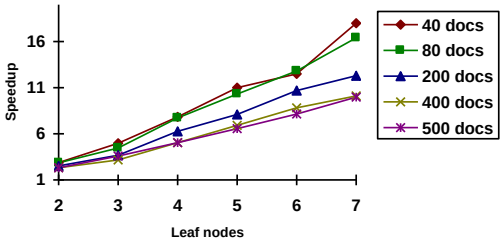


Figure 44. BASE1 [*DocId*]: index reorganisation speedup (position data)

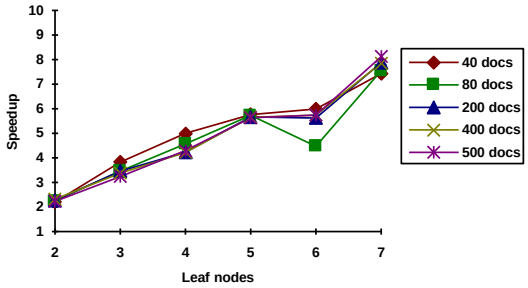


Figure 45. BASE1 [*TermId*]: index reorganisation speedup (postings only)

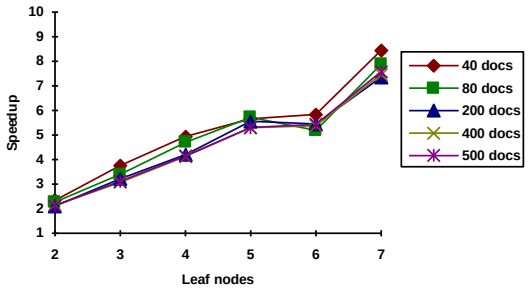


Figure 46. BASE1 [*TermId*]: index reorganisation speedup (position data)

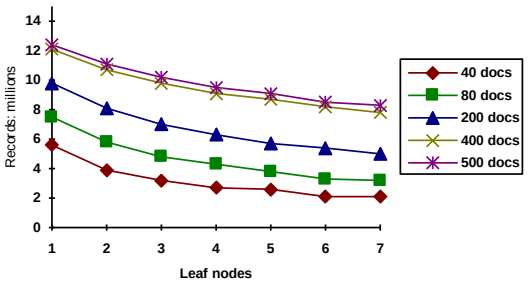


Figure 47. BASE1 [*DocId*]: millions of postings handled during index reorganisation

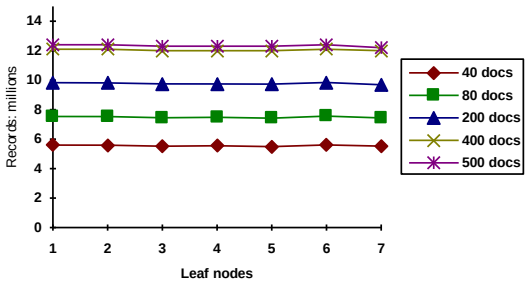


Figure 48. BASE1 [*TermId*]: millions of postings handled during index reorganisation

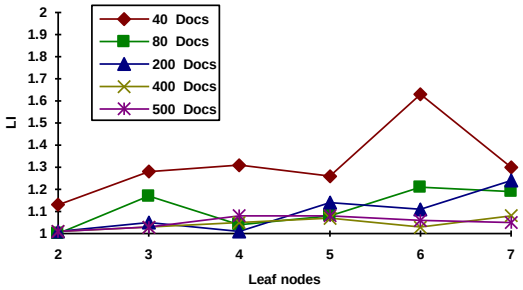


Figure 49. BASE1 [*DocId*]: index reorganisation load imbalance (postings only)

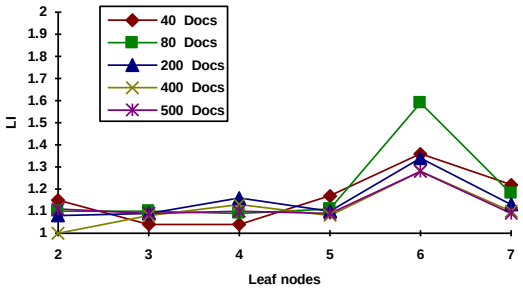


Figure 50. BASE1 [*TermId*]: index reorganisation load imbalance (postings only)

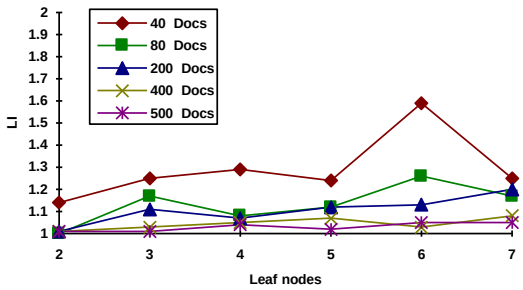


Figure 51. BASE1 [*DocId*]: index reorganisation load imbalance (position data)

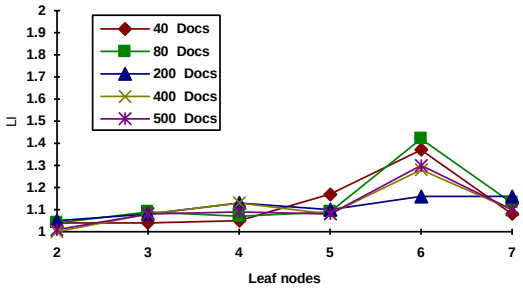


Figure 52. BASE1 [*TermId*]: index reorganisation load imbalance (position data)

Table I. Details of transaction sets used in experiments

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 II. BASE1/BASE10 [*DocId*]: index update results for update transactions

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 III. BASE1/BASE10 [*DocId*]: index update results for all transactions

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