Fast On-Line Index Construction by Geometric Partitioning

Nicholas Lester
RMIT University
Melbourne, Australia 3001
nml@cs.rmit.edu.au

Alistair Moffat
The University of Melbourne
Melbourne, Australia 3010
alistair@cs.mu.oz.au

Justin Zobel
RMIT University
Melbourne, Australia 3001
jz@cs.rmit.edu.au

ABSTRACT

Inverted index structures are the mainstay of modern text retrieval systems. They can be constructed quickly using off-line merge-based methods, and provide efficient support for a variety of querying modes. In this paper we examine the task of *on-line index construction* – that is, how to build an inverted index when the underlying data must be continuously queryable, and the documents must be indexed and available for search as soon they are inserted. When straightforward approaches are used, document insertions become increasingly expensive as the size of the database grows. This paper describes a mechanism based on controlled partitioning that can be adapted to suit different balances of insertion and querying operations, and is faster and scales better than previous methods. Using experiments on 100 GB of web data we demonstrate the efficiency of our methods in practice, showing that they dramatically reduce the cost of on-line index construction.

Categories and Subject Descriptors

H.3.1 [Information Storage and Retrieval]: Content Analysis and Indexing – *Indexing methods*; H.3.2 [Information Storage and Retrieval]: Information Storage – *File organization*; H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval – *Search process*; E.5 [Files]: Organization/structure

Keywords

Inverted file, inverted index, search engine.

1. INTRODUCTION

Inverted index structures are the mainstay of modern text retrieval systems. For example, an inverted index that stores document pointers only can be used for Boolean querying; an index that is augmented with $f_{d,t}$ within-document term frequencies can be used for ranked queries; and an index that additionally includes the locations within each document at which each term occurrence appears can be used for phrase querying. Witten et al. (1999) provide an introduction to these querying modes, and to inverted indexes.

CIKM'05, October 31–November 5, 2005, Bremen, Germany.

Inverted indexes can be constructed quickly using *off-line approaches*, in which one or more passes are made over a static set of input data, and, at the completion of the process, an index is available for querying. Witten et al. (1999) summarize several such off-line mechanisms; and in Heinz and Zobel (2003) describe a mechanism with improved performance. Off-line index construction algorithms are an efficient way of proceeding if a lag can be tolerated between when a document arrives in the system, and when it must be available to queries. For example, in systems indexing hundreds of megabytes of data, an hourly index build taking just a few minutes on a low-end computer is sufficient to ensure that all documents more than an hour old are accessible via the index. Systems in the gigabyte range might be similarly indexed daily, and in the terabyte range, weekly or monthly. A typical mode of operation in this case is for the new index to be constructed while queries are still being directed at the old; then for file pointers to be swapped, to route queries to the new index; and, finally, for the old index to be retired and its disk space reclaimed.

On the other hand, there are search environments in which even a small delay cannot be tolerated, and the index must always be queryable and up to date. In this paper we examine the corresponding task of *on-line index construction* – how to build an inverted index when the underlying data must be continuously queryable, and documents must be indexed for search as soon they arrive.

Making documents immediately accessible adds considerably to the complexity of index construction, and a range of tensions are introduced, with several quantities – including querying throughput, document insertion rate, and disk space – tradeable against each other. Here we describe a *geometric partitioning* mechanism that offers a range of tradeoffs between costs, and can be adapted to different balances of insertion and querying operations.

The principle of our new method is that the index is divided into a controlled number of partitions, where the capacities of the partitions form a geometric sequence. The presence of multiple partitions means that querying is slower than under a single-partition model, but as we demonstrate empirically, the overhead is not excessive. More significantly, the use of geometric partitions means that on-line index construction is faster and scales better than methods based on a single partitioning, while querying is faster than under other multiple-partition approaches. The new method leads to substantial practical gains; experiments with 100 GB of web data show that, compared to the alternative single-partition implementation, total construction throughput is more than three times greater.

2. INVERTED INDEX STRUCTURES

An inverted file index contains two main parts: a *vocabulary*, listing all the terms that appear in the document collection; and a

set of *inverted lists*, one per term. Each inverted list contains a sequence of *pointers* (also sometimes known as *postings*), together with a range of ancillary information, which can include within-document frequencies and a subsidiary list of positions within each document at which that term appears. A range of compression techniques have been developed for inverted lists (Witten et al., 1999; Scholer et al., 2002; Anh and Moffat, 2005), and, even if an index contains word positional information, it can typically be stored in around 25% of the space occupied by the original text. Index compression also reduces the time required for query evaluation.

The standard form of inverted index stores the pointers in each inverted list in document order, and is referred to as being *document sorted*. It is this index organization that we consider in this paper. Other index orderings are *frequency sorted* and *impact sorted*. None of the on-line mechanisms we describe in this paper apply to these other forms of index organization.

A retrieval system processes queries by examining the pointers for the query terms, and using them to calculate a similarity score that estimates the likelihood that the document matches the query. This process requires that all terms in the collection be indexed, with the exception of a small number of common terms, such as "the", and "of", that carry little information. Phrase queries can also be resolved via the index if it contains word positions. In this case the retrieval system treats each phrase in the query as a term, and infers an inverted list for it by combining the inverted lists for the component terms. Stop-words also need to be indexed for phrase queries to be efficiently resolved.

Inverted lists are typically stored on disk in a single contiguous extent or *partition*, meaning that once the vocabulary has been consulted, one disk seek and one disk read is required per query term.

The addition of a further document to an existing inverted index adds a new document pointer to a large number – potentially thousands – of inverted lists. Seeking on disk for each list update would be catastrophic, and in practical systems the disk costs are amortized across a series of updates. To this end, the inverted index in a dynamic retrieval system is stored in two parts: an in-memory component that provides an index for recently included documents; and an on-disk component, which is periodically combined with the in-memory part of the index in a *merging event*, and then written back to disk. This approach is effective because a typical series of documents has many common terms; the disk-based merging process can be sequential rather than random-access; all of the random-access operations can take place in main memory; and documents are searchable as soon as they are inserted. However, querying is now more complex, since the in-memory part of each terms' inverted list must be logically combined with the on-disk part.

The next section considers ways in which the merging process can be streamlined, and examines the cost of building a large index.

3. MERGE-BASED INDEX UPDATE

Three index maintenance algorithms have been explored in previous literature (Lester et al., 2004). They differ only in the manner in which the merge process is applied, but have significantly different costs. To analyse these costs, we assume that an index of n pointers is being constructed; that the in-memory index can hold b pointers, and hence that there are $\lfloor n/b \rfloor$ merging events; that it costs $n_1 + n_2$ steps to merge two indexes containing n_1 and n_2 pointers; and that the cost of preparing a buffer of b pointers for its first merge is $b \log v$ steps of computation, where v is the size of the vocabulary. The $\log v$ cost per pointer is for a search operation in a dictionary data structure; once the appropriate in-memory list has been identified, appending a new pointer to it takes constant time.

To quantify some of the calculations, typical values for the two key values are $n \approx 10^{10}$, and $b \approx 10^8$. An index of ten billion pointers might arise, for example, in a collection of 100 GB; and one hundred million pointers stored in the in-memory part of the index would require approximately 500 MB of main memory, and giving rise to about 200 MB of compressed postings on disk. Similarly, we would expect v to be of the order of 3×10^7 for the entire collection, corresponding to seeing one new word approximately every 300 term appearances.

Rebuilding

The simplest strategy is to entirely rebuild the on-disk index from the stored collection whenever the in-memory part of the index exceeds b pointers. Despite the obvious disadvantages of this *rebuild* strategy, it is not unreasonable for small collections with low update frequency. For example, Lester et al. (2004) give experimental results showing that this technique does work moderately well in some situations.

At each index rebuild all previous information is discarded, and the data to date completely reprocessed. The total cost is thus

$$\begin{aligned} cost_{rebuild}(n, b, v) &= \sum_{i=1}^{n/b} (ib + ib \log v) \\ &\leq b(1 + \log v) \sum_{i=1}^{n/b} i \\ &\approx \frac{n^2 \log v}{2b} . \end{aligned}$$

That is, for any fixed value of b, rebuild requires time that grows (at least) quadratically. For the values of n, b, and v hypothesized above, the cost amounts to some 17.1 trillion operations.

Remerge update

The second merging strategy avoids re-processing pointer lists that are already ordered, and adds *bufferloads* to the on-disk index, appending to each inverted list. At each merge event, the entire on-disk index is read sequentially and written in extended form to a new location, with the pointers from the in-memory part of the index inserted as appropriate. When the output is completed, the original index files are freed, and querying transferred over to the new index. This *remerge* approach has the disadvantage that the entire index must be processed every time the in-memory and on-disk components are combined. Unless care is taken, it also means that the peak disk usage is twice the cost of storing the index (Clarke and Cormack, 1995; Moffat and Bell, 1995). On the other hand, the remerge strategy is significantly faster than the rebuild approach.

The merge between the in-memory postings and the on-disk postings, both of which are sorted, incurs a cost proportional to the total number of postings in the resulting index. In addition, each bufferload of pointers must be sorted. Over the n/b merging events the cost is thus

$$\begin{aligned} cost_{remerge}(n, b, v) &= \sum_{i=1}^{n/b} (b \log v + ib) \\ &= \frac{n}{b}(b \log v) + b \sum_{i=1}^{n/b} i \\ &\approx n \log v + \frac{n^2}{2b} . \end{aligned}$$

For the presumed values of n, b, and v, this indicates a cost of approximately 0.8 trillion operations to build an index, significantly better than the number of steps required by the rebuild approach.

In-place update

The third strategy, referred to as *inplace*, involves minimizing the change to the index at each stage by, whenever possible, writing new postings at the end of the existing lists. Each list must be periodically moved to unused space so that there is enough room for the new postings; to avoid excessive numbers of moves, it is necessary to over-allocate by a constant factor the space used for the on-disk inverted lists. That is, during each merging event, pointers from the in-memory index are transferred into the free space at the end of the term's on-disk inverted list. If insufficient free space is available at the end of the on-disk list, the combined list is copied to a new location, and again over-allocated. Variations include keeping short postings lists within the vocabulary structure (Cutting and Pedersen, 1990); keeping short lists within fixed-size "bucket" structures (Shoens et al., 1994); and predictive over-allocation for long postings lists to reduce relocations and discontinuity (Tomasic et al., 1994; Shieh and Chung, 2003).

The staggered growth of inverted lists introduces a free-space management problem. Space management of large binary objects, such as image data, is examined by a number of authors (Biliris, 1992a,b; Carey et al., 1986, 1989; Lehman and Lindsay, 1989), although it should also be noted that text databases present different problems than other types of binary data.

To analyze the execution cost of building the index, suppose that each list is over-allocated by a fixed factor of $r > 1$. When the list for some term t contains f_t pointers and is at its current capacity, and a $f_t + 1$st posting is to be added to it during the merge event, the list is first extended to be $\lceil f_t r \rceil$ postings, and only then is the new one added. Consider the set of pointers in the just-extended list. One of them – just appended – has never taken part in a list extension operation. A further $f_t - f_t/r$ items have just been copied for the first time; $f_t/r - f_t/r^2$ items have just been moved for the second time; and so on. The average number of times each of those $f_t + 1$ pointers have been moved is given by

$$\begin{aligned} & \frac{1}{f_t+1}\left(1-\frac{1}{r}\right)\left(\frac{f_t}{1}+\frac{2f_t}{r}+\frac{3f_t}{r^2}+\cdots\right) \\ \approx\ & \left(1-\frac{1}{r}\right)\sum_{i=0}^{\infty}\frac{i+1}{r^i} \\ =\ & \frac{r}{r-1}\,. \end{aligned}$$

For example, when $r = 1.25$, just after a list is resized, the pointers in it have been moved five times on average. This amortized limit is independent of the size f_t of the list, and applies to all lists just after they have been extended, which is when the average number of moves is at its highest. The expression above is thus an upper bound on the per-pointer cost of constructing the whole inverted index.

The cost of adding every item to the list the first time must be added to the copying cost, and also the $b \log v$ cost of preparing each of the n/b the bufferloads before each merge event takes place. Hence the total number of operations required to process n pointers in bufferloads of size b is given by

$$cost_{\textit{inplace}}(n, b, v) = n\left(\log v + \frac{2r-1}{r-1}\right)\,.$$

When n, b, and v are as suggested at the beginning of this section, and $r = 1.25$ is used, the total number of operations is a little over 0.3 trillion.

The inplace method does, however, have two distinct disadvantages that partly or completely negate this smaller number of operations. First is the memory overhead. Around 60% of the over-allocated space is always vacant. That is, when $r = 1.25$, around 15% of the disk space allocated to the lists in index is unused. There will also be external fragmentation not taken into account in this computation, caused by the chaotic overall sequence of list resizings. This could easily add a further 5%–25% space overhead, depending on the exact disk storage management strategy used.

The second problem is that processing speed is not as fast as the analysis above would suggest – it counted only data movements, and in the inplace mechanism a non-trivial amount of random-access processing of the index file is required during each merge event. In contrast, the operations performed in the remerge method are strictly sequential.

As a minimum, all of the terms that have occurred in each bufferload generate a disk operation. With perhaps 10^6 distinct terms expected in each bufferload (that is, one per one hundred pointers, and higher than the steady-state rate of one per three hundred pointers because of start-up considerations), a minimum of perhaps 10^8 disk accesses is required over all bufferloads. Using a very crude relativity of one disk seek corresponding to 10^6 sequential data movements, handling each term in each bufferload thus adds a further $10^{12} = 1$ trillion operations to the execution cost.

Lester et al. (2004) evaluated the three mechanisms described in this and the previous two section. Their experiments confirmed that the remerge scheme is better than the inplace scheme in almost all practical scenarios, despite remerge processing the entire index during every merging event.

Multiple partitions

One of the primary reasons that maintenance algorithms are inefficient is that they keep a single, contiguous, inverted list per term. On the other hand, sort-based bulk-construction algorithms are not constrained in this regard, and make use of multi-way merging strategies to reduce the number of times each pointer is handled. For example, Heinz and Zobel (2003) describe a bulk-construction algorithm in which the dominant operation is to write a bufferload of pointers to a separate file. The cost of each such operation is independent of the total amount of data being indexed, unlike the methods analyzed above.

Tomasic et al. (1994) describe an index maintenance strategy that avoids re-processing data by creating new discontiguous list fragments as inverted lists grow. Their scheme is like the inplace scheme, except that when a list outgrows its allocation, the old list is left unchanged and a new fragment is allocated in order to hold additional postings. They do not use over-allocation for any but the first fragment, so their algorithm creates approximately one fragment per combination event.

It is straightforward to alter the Tomasic et al. algorithm to include predictive over-allocation, but the approach still results in each inverted list being spread across a number of fragments that grow linearly in the size of the collection, and query processing times suffer accordingly.

Assuming that the index for each bufferload is written to memory and then linked from the previous partition of the index lists, the processing time for the Tomasic et al. approach is

$$cost_{\textit{Tomasic}}(n, b, v) \approx n\,(1 + \log v)$$

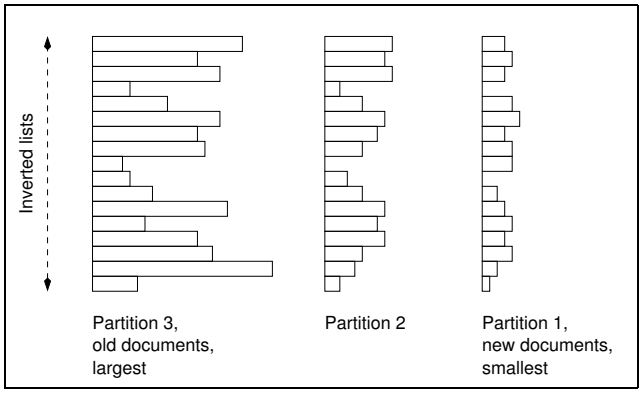

Figure 1: Geometrically partitioned index structure, with three levels. The oldest index pointers are in the lowest, largest partition, which in this example is at level 3. The vocabulary, not shown in the figure, includes three pointers with each term's entry. Some terms are absent from some partitions.

plus the time needed to create the chains of pointers that thread the structure together.

Query costs scale badly in this approach, or with lists represented as linked chains of fixed-size objects (Brown et al., 1994). For the values of n and b supposed in the example, querying would entail as many as 100 disk seeks per query term, an unpalatable cost.

4. GEOMETRIC PARTITIONING

We propose a scheme that blends the remerge method described in Section 3, and the too-many-fragments approach of Tomasic et al. The key idea is to break the index into a tightly controlled number of partitions. Limiting the number of partitions means that as the collection grows there must continue to be merging events, but they can be handled rather more strategically than before, and the result is a net saving in processing costs. (Because each index list is in multiple parts, querying is slower than with single-partition lists. In Section 5 we quantify the amount of that degradation.)

At any given point, the index is the concatenation of the set of partitions, each of which is a partial index for a contiguous subset of the documents in the collection. We also impose an ordering on partition sizes, requiring that the partition containing the most recently added documents be the smallest. Similarly, the first documents in the collection are indexed via the largest partition. Figure 1 shows an example arrangement in which the on-disk part of the index is split into three partitions.

The vocabulary entry for each term records multiple disk addresses, one for each partition. The locations of all the index partitions for a term are then available at the cost of a single disk read, although the vocabulary will typically be slightly larger as a result. When queries are being processed, each of those partitions on disk is retrieved and processed, as in the approach of Tomasic et al. The difference in our approach is that we ensure, via periodic mergings, that the number of partitions does not grow excessively.

Consider what happens when an in-memory bufferload of postings is to be transferred to disk. That bufferload can be merged with any one of the partitions, or indeed, with any combination of them. The key issue to be addressed is how best to manage the sequence of mergings so as to minimize the total merging cost, without allowing the number of partitions to grow excessively.

The linear cost of each merging step means that, for it to be relatively efficient, the two lists should not differ significantly in size. To this end, we introduce a key parameter r, and use it to define the capacity of the partitions: the limit to the number of pointers in one partition is r times the limit for the next. In particular, if a bufferload contains b pointers, we require that the first partial index not exceed $(r-1)b$ pointers; the second partial index not contain more than $(r-1)rb$ pointers; and, in general, the kth partial index not more than $(r-1)r^{k-1}b$ pointers. In addition, r also specifies a lower bound on the size of each partition – at level k the partition is either empty, or contains at least $r^{k-1}b$ pointers.

In combination, these two constraints ensure that, when a merge of two partitions at adjacent levels takes place, the combined output is not more than r times bigger than the smaller of the two input partitions, and is at least $r/(r-1)$ times bigger than the larger. As is demonstrated shortly, this relationship allows useful bounds to be established on the total cost of all mergings.

Hierarchical merging

The limits on the capacity of each partition give rise to a natural sequence of hierarchical merges that follows the radix-r representation of the number of bufferloads that have been merged to date. Suppose, for example, that $r = 3$, and, as before, the stream of arriving documents is processed in fixed bufferloads of b documents.

The first bufferload of pointers is placed, without change, into partition 1. The second bufferload of pointers can be merged with the first, still in partition 1, to make a partition of $2b$ pointers. But the third bufferload of pointers cannot be merged into partition 1, because doing so would violate the $(r-1)b = 2b$ limit on partition 1. Instead, the result of the merge is placed in partition 2, and partition 1 is cleared.

The fourth bufferload of pointers must be placed in partition 1, because it cannot be merged into partition 2. The fifth joins it, and then the sixth bufferload triggers a three-way merge, to make a partition containing $6b$ pointers in the second partition. Figure 2 continues this example, and shows how the concatenation of three more bufferloads of pointers from the in-memory part of the index leads to a single index of $9b$ pointers in the third partition.

Analysis

Within each partition the index sizes follow a cyclic pattern that is determined by the radix r. For example, in Figure 2, the "Partition 2" column cycles through sizes 0, 3, 6, and then repeats. In general, the kth partition of an index built with radix r cycles through the sequence $0, r^{k-1}, 2r^{k-1}, \ldots, (r-1)r^{k-1}$. Over one full cycle this sequence sums to

$$\frac{(r-1)r}{2} r^{k-1} = \frac{(r-1)r^k}{2} .$$

We will make use of this quantity shortly.

Each of the numbers in the cycle is exactly the cost of forming the corresponding partition, since it is the sum of the sizes of the segments that were joined together to make that partition. For example, to build an index of size $9b$ pointers with $r = 3$, the total merging cost is the sum of all of the partition sizes in Figure 2, which amounts to three cycles through partition 1 (total cost: $9b$), one cycle through partition 2 (total cost: $9b$), and a single merge of cost $9b$ in partition 3, for a total of $27b$ units.

For index of n pointers in total, the merging pattern has $\lfloor n/b \rfloor$ rows. Hence, in the ith column there will be at most $\lfloor n/b \rfloor / r^i$ full cycles of the merging pattern; furthermore, the average merg-

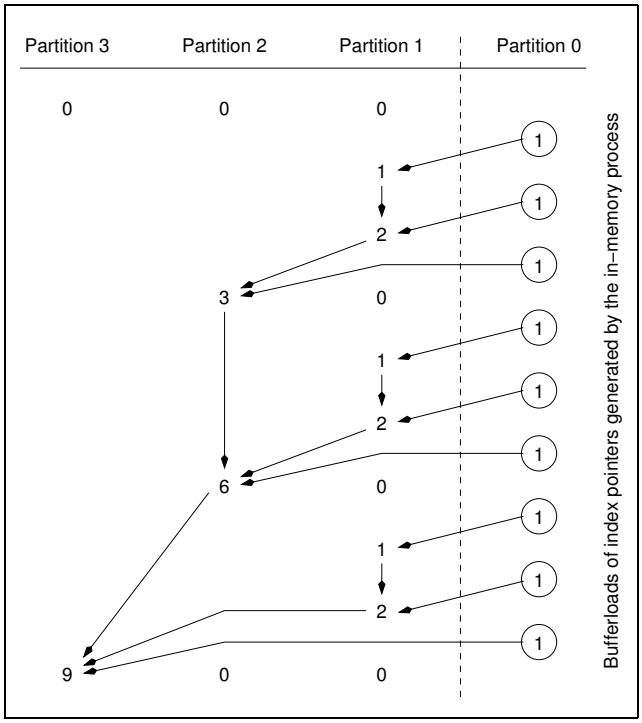

Figure 2: The merging pattern established when $r = 3$. After nine bufferloads have been generated by the in-memory part of the indexing process, the first index is placed into partition 3. All numbers listed represent multiples of b, the size of each bufferload.

ing cost in any partial cycles that have taken place is less than the average cost over the completed cycles.

For fixed values of n, b, and r, the number of partitions (columns) p required is

$$p = 1 + \lfloor \log_r(n/b) \rfloor \approx 0.5 + \frac{\log(n/b)}{\log r} .$$

Summing over all columns and all cycles of the merging pattern (rows), the total cost of the merging stages is thus the constant multiplier b times

$$\sum_{i=1}^{p} \frac{\lfloor n/b \rfloor}{r^i} \frac{(r-1)r^i}{2} \approx \frac{(r-1)n}{2b} \left(0.5 + \frac{\log(n/b)}{\log r} \right) .$$

Multiplying by the scale factor b, and adding in the $n(1 + \log v)$ total cost of the in-memory stages, gives

$$\begin{aligned} & cost_{geom}(n, b, v, r) \\ &= n \left(1 + \log v + \frac{r-1}{2} \left(0.5 + \frac{\log(n/b)}{\log r} \right) \right) . \end{aligned}$$

The same total number of pointers are in the index at any given moment, regardless of the partition arrangement. But breaking each inverted list into partitions does introduce additional disk seek operations, and slows overall query throughput rates, since all non-zero partitions need to be fetched. Over the r partition sizes in each cycle, there is a $1/r$ chance of that partition being empty at any given time. Multiplying by the number of partitions p means that, per query term, the disk access cost is approximately

$$\frac{r-1}{r} \left(0.5 + \frac{\log(n/b)}{\log r} \right) .$$

r	Build cost	Access cost
2	0.29	3.6
3	0.31	3.1
4	0.32	2.9
6	0.34	2.6
8	0.35	2.4
12	0.39	2.2
16	0.42	2.0

Table 1: Index construction cost (trillions of operations, calculated) and the expected number of disk accesses per query term (calculated), for $n = 10^{10}$, $b = 10^8$, $v = 3 \times 10^7$, and different values of r.

Table 1 quantifies, for several values of r, the total index build cost (in trillions of operations, calculated according to $cost_{geom}(n, b, v, r)$) and the expected number of disk accesses per term (using the formula above), for the values $n = 10^{10}$, $b = 10^8$, and $v = 3 \times 10^7$.

The multiple-partitions method of Tomasic et al. (1994) can be seen as an extreme form of this method. Pointers are never moved, so index creation cost is linear, but querying costs are high.

There are two issues in the analysis that require elaboration.

The first is the cost attributed to a multi-way merge. In Figure 2, for example, it was assumed that the three-way merge to create the partitions of size six and nine cost six and nine units of work, respectively. More generally, the analysis assumes that a k-way merge between objects of size $n_1, n_2, \ldots, n_k$ takes time $\sum_{i=1}^{k} n_k$.

When merging lists by a sort key an additional factor of $\log_2 k$ is required, to account for the cost of computing, at each step in the merge, the next smallest key among the k candidates. In the merging operation being performed here, data movements dominate comparisons, because the inverted lists get proportionately longer at each merge. Put another way, the comparison cost of each merge is $(n'_1 + \cdots + n'_k) \log k$, where n'_i (the number of distinct terms in the ith input to the merge) is a steadily declining fraction of n_i (the total number of pointers in the ith input to the merge). On this basis, we believe the analysis to be fair.

Another issue that has been simplified in this analysis is that of processing the vocabulary. The vocabulary grows in size as bufferloads are incorporated into the index, and is processed sequentially in its entirety at every merge, even when the merge is in partition 1 to carry out $b + b \rightarrow 2b$. Indeed, the vocabulary is approximately proportional in size to the number of pointers in the index, reflecting the observation that new words appear at a steady rate no matter how large the collection has already grown (Williams and Zobel, 2005). In our experiments, detailed below, new words were encountered at a rate of one per three hundred pointers, leading to the final value $v = 3 \times 10^7$ used in the previous calculations.

At the time of the kth merge the vocabulary can be assumed to contain $bk/300$ entries. Supposing also that each vocabulary entry occupies ten times the storage of a single pointer (the vocabulary entry stores a string, disk pointers into the partitions, and the size in each partition of the corresponding list of pointers, whereas as a pointer is typically 3–5 bytes long), the cost of sequentially pro-

cessing the vocabulary is thus

$$\sum_{k=1}^{n/b} \frac{10kb}{300} \approx \frac{n^2}{60b}\,.$$

For n and b as discussed, this amounts to less that 0.02 trillion operations, and does not greatly affect the costs shown in Table 1.

On the other hand, when n (and thus v) are large, the quadratic nature of this component of the running time means that it will eventually dominate. If there is a risk of that happening, the vocabulary data structure should be modified to one that supports merging of specific pages rather than simple sequential merging of the whole structure.

Varying the radix

The best choice of r depends on the balance of operations. Table 1 suggests that use of an overly small value of r is likely to harm query costs, and should be avoided when the operation mix is dominated by queries; similarly, if the operation mix is dominated by insertions, smaller values of r are to be preferred.

It is also possible to consider the number of partitions p to be the fixed quantity, and determine r accordingly, so as to never require more than p partitions. Doing so makes the seeks-per-term part of the querying cost largely independent of n, at the expense of slowly increasing per-insertion times.

When p is fixed, and the index restricted to not more than p levels, the ratio r must be such that

$$\frac{n}{b} \le 1 + r + r^2 + \cdots + r^p = \frac{r^{p+1}-1}{r-1}\,.$$

Setting

$$r = \left\lceil \left(\frac{n}{b}\right)^{1/p} \right\rceil = \left\lceil \sqrt[p]{\frac{n}{b}} \right\rceil$$

is sufficient to meet the requirement, and suggests an approach in which p is fixed, and r is varied as necessary as the index grows.

Figure 3 shows the merging sequence for $p = 2$. As the tenth bufferload of text is processed, r is incremented to $\lceil\sqrt{10}\,\rceil = 4$. The next few sizes of the second partition are 15, 20 (because r is increased to 5), 25, 31, and then 38. The remerge strategy of Section 3 is thus simply a special case of this strategy, that of $p = 1$, with every bufferload of pointers merged into a single partition.

The execution cost of this variant scheme is bounded above by

$$\begin{aligned} &cost_{geom}(n, b, v, r) \\ &\quad = \quad cost_{geom}\left(n, b, v, (n/b)^{1/p}\right) \\ &\quad = \quad n(1+\log v) + n \cdot \frac{(n/b)^{1/p} - 1}{2} \cdot (0.5 + p) \\ &\quad \approx \quad n(1+\log v) + \frac{p n^{1+1/p}}{2b^{1/p}}\,, \end{aligned}$$

which is asymptotically dominated by the $O(n^{1+1/p})$ term. However, for typical values of n and b the first term is larger numerically, and the most important component of the running time. Table 2 shows the actual number of operations required to build an index for the hypothesized values of n and b, for various values of the bound p, calculated by simulating the "fixed p, varying r" merging process through a total of $n/b = 100$ bufferloads and summing the costs of the merges performed.

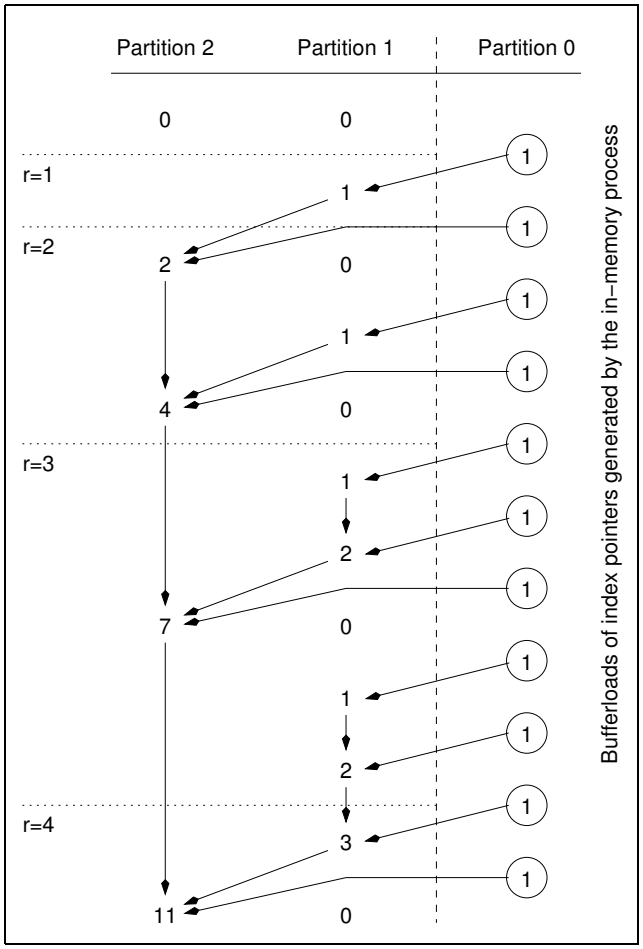

Figure 3: The merging pattern established when $p = 2$ and r is varied. All numbers listed represent multiples of b, the size of each bufferload.

5. EXPERIMENTS

To validate the analyses, we experimented with the `wt100g` web-based document collection (Hawking et al., 1999), and measured index construction time and querying time for a range of on-line and off-line indexing techniques. All experiments were performed on a dual-processor Intel Pentium 4 2.8 GHz machine with hyper-threading turned on, but employing only a single processor. The experimental machine had 2 GB of RAM and was under light load, with no significant other processes running at the time of experiment. Times presented are elapsed times, including all parsing, indexing, and list merging phases. In each experiment the construction process accumulated 80×10^6 pointers in memory, corresponding to approximately 200 MB of compressed index, prior to each merge event. On the `wt100g` collection this gave rise to 99 bufferloads of postings needing processing.

Index construction times for two different variants of the on-line partitioned approach are shown in Figure 4, and compared to those of the off-line construction method of Heinz and Zobel (2003), and the remerge on-line method. Only the on-line methods can support querying at intermediate stages of construction, and the line labelled "bulk construction" reflects the cost of constructing an index of that size off-line, rather than the cost of incrementally build-

p	Final r	Build cost
2	10	0.34
3	5	0.30
4	4	0.29
5	3	0.29

Table 2: Index construction cost (trillions of operations) for $n = 10^{10}$ and $b = 10^8$, and different fixed values of p. The final value of r in each case is also shown.

ing the index. Times for the rebuild and inplace methods are not shown, but are considerably slower than remerge for this combination of data and buffer size.

The relationships between the methods plotted in Figure 4 are as expected, with the $r = 3$ version leading to a slightly shorter execution time than the $p = 2$ variant. The super-linear growth rate is also as expected. The bulk construction mechanism generates n/b bufferloads of pointers at total cost $O(n \log v)$, and then merges them all together in a single $\lceil n/b \rceil$-way merge, in (by the merging cost model assumed in this paper) $O(n)$ time. On the other hand, for fixed values b, the cost of the partitioned method is $O(n \log_r n)$; and when $p = 2$ forces $r = \sqrt{n}$, this equates to $O(n^{1.5})$ time.

Figure 5 shows the effect of the partitioned index construction scheme on querying efficiency when $r = 3$, and when $p = 2$. The upper graph shows the number of non-zero partitions at each stage in the construction process, essentially counting the number of non-zero digits in the base-r representation of k, the number of bufferloads processed to date. The lower graph shows the time taken to resolve 10,000 queries against the index at that stage of its construction. The 10,000 queries were taken from the start of the standard Excite query log (Spink et al., 2001). Preliminary experiments not described here demonstrated that 10,000 queries were enough to get reasonably stable measurements of execution time.

The query times in Figure 5 show that the overhead arising from the use of multiple partitions is definite, but not excessive. The greatest overhead, when there are four partitions, is approximately 40%; but on the $p = 2$ line the overhead is not more than 20%. It probably represents the better compromise between index construction time and querying times, for this data at least.

There is a strong correlation in Figure 5 between the query overhead (in the lower graph) and the number of index partitions (in the upper graph), suggesting that the overhead is primarily a result of the partitioning of index lists rather than any other factors, and that our models are capable of predicting the level of query overhead relatively accurately. Similar behavior is also present in a less pronounced manner in Figure 4, where the slight upward steps represent significant whole-of-index merging events.

6. CONCLUSIONS

We propose a mechanism for on-line index construction for text databases that is based on the principle of dividing the index into a small number of partitions of geometrically increasing size. In contrast to update mechanisms for standard contiguous representation of inverted indexes, construction costs are significantly reduced, and more scalable.

Our experiments have quantified the effect in practice of representing an index as geometrically sized partitions. As predicted by our analysis, index construction is much more efficient; the time required to build an index for a 100 GB collection is reduced by a factor of around four compared to a comparable implementation

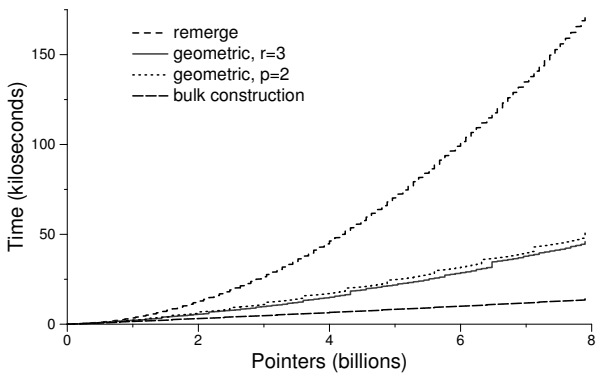

Figure 4: Time required to build an index for the `wt100g` collection using 99 bufferloads each containing approximately 200 MB of pointers. The line denoted "bulk construction" shows the comparable cost of an off-line index construction algorithm for that amount of data.

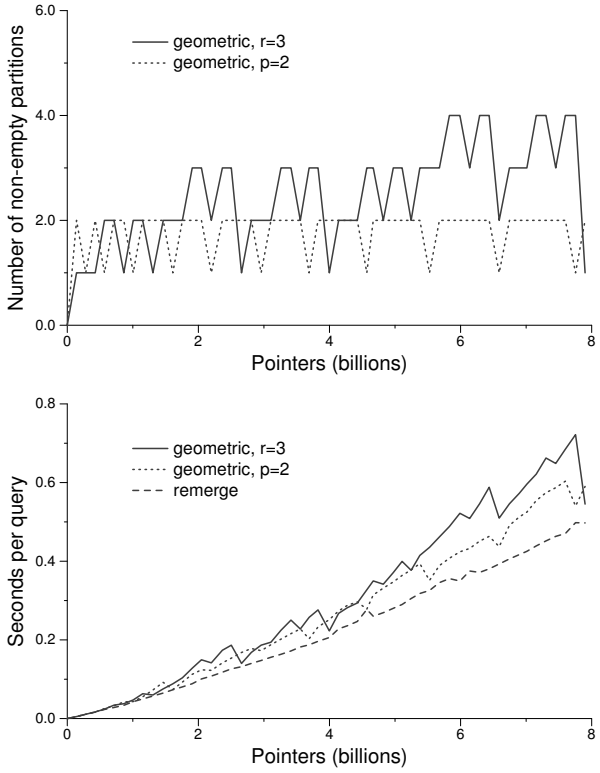

Figure 5: Query times on partially built indexes for the `wt100g` using 54 bufferloads each containing approximately 355 MB of pointers, using $r = 3$ and $p = 2$ constraints. The upper graph shows the number of non-zero partitions after each bufferload is incorporated, the lower graph shows the measured per-query cost of evaluating 10,000 Excite queries. Values for querying time and number of partitions in use are plotted after each bufferload of pointers is incorporated into the on-disk part of the index.

of the remerge approach. The results also show that the relative gains increase with collection size: the time to add the last gigabyte is around an hour for remerge, but just nine minutes for the geometric approach. On the other hand, construction times are still considerably higher than for off-line index construction.

The main disadvantage of multiple partitions is that querying is slower. But by limiting the number of partitions, the degradation in query time is modest; our experiments show that with $p = 2$, queries on average take around 20% longer. As the number of partitions can be controlled either indirectly through the choice of radix r, or explicitly via a fixed limit p, a retrieval system can be tuned to the mix of querying and update operations that is anticipated.

Thus, by restricting the way in which the index partitions grow in size, we have been able to bound the total cost of the index construction process, and have also bounded the extra cost that arises in query processing. Our work shows that on-line methods offer an attractive compromise between construction costs, querying costs, and access immediacy.

We note that Büttcher and Clarke (2005) have independently described a method similar to the work presented here.

Acknowledgments

This project was supported by the Australian Research Council.

References

V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, January 2005. Source code available from `www.cs.mu.oz.au/~alistair/carry/`.

A. Biliris. An efficient database storage structure for large dynamic objects. In F. Golshani, editor, *Proc. IEEE Int. Conf. on Data Engineering*, pages 301–308, Tempe, Arizona, February 1992a. IEEE Computer Society. ISBN 0-8186-2545-7.

A. Biliris. The performance of three database storage structures for managing large objects. In M. Stonebraker, editor, *Proc. ACM-SIGMOD Int. Conf. on the Management of Data*, pages 276–285, San Diego, California, June 1992b.

E. W. Brown, J. P. Callan, and W. B. Croft. Fast incremental indexing for full-text information retrieval. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. Int. Conf. on Very Large Databases*, pages 192–202, Santiago, Chile, September 1994.

S. Büttcher and C. L. A. Clarke. Indexing time vs. query time trade-offs in dynamic information retrieval systems. In N. Fuhr, H.-J. Schek, and A. Chowdhury, editors, *Proc. ACM CIKM Int. Conf. on Information and Knowledge Management*, Bremen, Germany, 2005.

M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *Proc. Int. Conf. on Very Large Databases*, pages 91–100, Kyoto, Japan, August 1986. Morgan Kaufmann. ISBN 0-934613-18-4.

M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Storage management for objects in EXODUS. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 341–369. Addison-Wesley Longman, New York, 1989.

C. L. A. Clarke and G. V. Cormack. Dynamic inverted indexes for a distributed full-text retrieval system. Technical Report MT-95-01, Department of Computer Science, University of Waterloo, Waterloo, Canada, 1995. MultiText Project Technical Report.

D. R. Cutting and J. O. Pedersen. Optimizations for dynamic inverted index maintenance. In J.-Luc Vidick, editor, *Proc. ACM-SIGIR Int. Conf. on Research and Development in Information Retrieval*, pages 405–411, Brussels, Belgium, September 1990. ACM. ISBN 0-89791-408-2.

D. Hawking, N. Craswell, and P. Thistlewaite. Overview of TREC-7 very large collection track. In E. M. Voorhees and D. K. Harman, editors, *The Eighth Tert REtrieval Conference (TREC-8)*, pages 91–104, Gaithersburg, MD, 1999. National Institute of Standards and Technology Special Publication 500-246.

S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *Jour. of the American Society for Information Science and Technology*, 54(8):713–729, 2003.

T. J. Lehman and B. G. Lindsay. The Starburst long field manager. In P. M. G. Apers and G. Wiederhold, editors, *Proc. Int. Conf. on Very Large Databases*, pages 375–383, Amsterdam, The Netherlands, August 1989. ISBN 1-55860-101-5.

N. Lester, J. Zobel, and H.E. Williams. In-place versus re-build versus re-merge: Index maintenance strategies for text retrieval systems. In V. Estivill-Castro, editor, *Proc. Australasian Computer Science Conf.*, pages 15–22, January 2004.

A. Moffat and T. A. H. Bell. In situ generation of compressed inverted files. *Journal of the American Society of Information Science*, 46(7):537–550, 1995.

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

W.-Y. Shieh and C.-P. Chung. A statistics-based approach to incrementally update inverted files. In H. R. Arabnia, editor, *Proc. Int. Conf. on Information and Knowledge Engineering*, pages 38–43, Las Vegas, Nevada, June 2003. CSREA Press.

K. Shoens, A. Tomasic, and H. García-Molina. Synthetic workload performance analysis of incremental updates. In W. B. Croft and C. J. van Rijsbergen, editors, *Proc. International ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 329–338, Dublin, Ireland, July 1994.

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

A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *Proc. ACM-SIGMOD Int. Conf. on the Management of Data*, pages 289–300, Minneapolis, Minnesota, May 1994. ACM.

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

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