Efficient Online Index Construction for Text Databases

NICHOLAS LESTER
RMIT University, and Microsoft Corporation
ALISTAIR MOFFAT
The University of Melbourne
and
JUSTIN ZOBEL
RMIT University

Inverted index structures are a core element of current text retrieval systems. They can be constructed quickly using offline 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. However, there are search environments in which even a small delay in timeliness cannot be tolerated, and the index must always be queryable and up to date. Here we describe and analyze a *geometric partitioning* mechanism for online index construction that provides a range of tradeoffs between costs, and can be adapted to different balances of insertion and querying operations. Detailed experimental results are provided that show the extent of these tradeoffs, and that these new methods can yield substantial savings in online indexing costs.

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.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation (efficiency and effectiveness)*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Index construction, index update, search engines, text indexing

This article incorporates and extends "Fast on-line index construction via geometric partitioning" by N. Lester, A. Moffat, and J. Zobel, in *Proceedings of the International Conference on Information and Knowledge Management* (CIKM), 2005, 776–783.
This article was completed while N. Lester was employed by RMIT University, with support from the Australian Research Council.
Authors' addresses: N. Lester, One Microsoft Way, Redmond, WA; email: nlester@microsoft.com; A. Moffat, Department of Computer Science and Software Engineering, The University of Melbourne, Victoria 3010, Australia; email: alistair@csse.unimelb.edu.au; J. Zobel, School of Computer Science and Information Technology, RMIT University, Victoria 3001, Australia; email: jz@cs.rmit.edu.au.

ACM Reference Format:
Lester, N., Moffat, A., and Zpbel, J. 2008. Efficient online index construction for text databases. ACM Trans. Datab. Syst. 33, 3, Article 19 (August 2008), 33 pages. DOI = 10.1145/1386118.1386125 http://doi.acm.org/10.1145/1386118.1386125

1. INTRODUCTION

Inverted index structures are a core element of current text retrieval systems. An inverted index that stores document pointers can be used alone 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. Zobel and Moffat [2006] and Witten et al. [1999] provide an introduction to these querying modes and to inverted indexes.

Inverted indexes can be constructed quickly using *offline 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. Zobel and Moffat [2006] summarize several such offline mechanisms. Offline 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 in which hundreds of megabytes of data are indexed, 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 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 one 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 in timeliness cannot be tolerated, and the index must always be queryable and up to date. In this article we examine the corresponding task of *online 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. We show, moreover, that the concept of partitioning can be applied to both the inverted lists and the B^+-tree [Ramakrishnan and Gehrke 2003] that maintains the vocabulary.

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 online index construction is faster and scales better than methods based on a single partition, while querying is faster than under other multiple-partition approaches. The new method leads to substantial practical gains; experiments with 426 GB of Web data show that, compared to the alternative single-partition implementation, construction speed is several times faster. Our experiments also demonstrate that the reduction in asymptotic cost of incremental index construction offered by geometric partitioning can be achieved in practice.

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 known as *postings*), together with a range of ancillary information, which can include within-document frequencies, together with a subsidiary list of positions within each document at which that term appears. A range of compression techniques have been developed for inverted lists [Scholer et al. 2002; Anh and Moffat 2006; Zobel and Moffat 2006], 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, referred to as being *document-sorted*. It is this index organization that we consider in this article. Other index orderings are *frequency-sorted* and *impact-sorted*. None of the online mechanisms we describe in this article apply to these other forms of index organization. Inverted lists are typically stored on disk in a single contiguous extent or *partition*, meaning that once the vocabulary has been consulted, one logical disk seek and one logical disk read are required per query term.[1]

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 possible exception of a small number of common terms 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 lists for the component terms. Stop-words also need to be indexed for phrase queries to be efficiently resolved.

[1]Note that in many operating systems, Unix included, logical disk operations may be translated into multiple physical disk operations as index blocks (i-nodes) and the like are processed. In all of the work reported here, we presume that the operating system makes effective use of the underlying physical disk, and that sequential logical disk operations are translated into physical disk operations with a negligible amount of overhead. The alternative would be for us to implement our own raw file system, which is appropriate for a commercial setting, but not a research one.

Adding just one new document to an existing index adds a new pointer to each of a large number—potentially thousands—of inverted lists. Seeking on disk for each list update would be catastrophic, so 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 that is periodically combined with the in-memory *bufferload* of pointers 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; and because all of the random-access operations can take place in main memory. 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.

In order to efficiently access inverted lists while resolving queries, information retrieval systems also maintain a central vocabulary. This structure maps each indexed term to an inverted list, and also requires update when the contents of the inverted lists change.

We assume that the vocabulary is stored using a B^+-tree, as this data structure scales beyond the capacity of main memory, and is commonly used in practice. B^+-tree update is a well-explored problem [Bayer 1972; Knuth 1973; Comer 1979; Graefe 2006].

3. INDEX CONSTRUCTION TECHNIQUES

To analyze the costs associated with index construction, we assume that an index of n pointers is being built; that the in-memory index can hold b pointers, and hence that there are $\lceil n/b \rceil$ merging events; and that the initial cost of inverting each bufferload of b pointers is $b \log b$ steps of computation. The $\log b$ cost per pointer is for a search operation in a dictionary data structure containing not more than b items; once the appropriate in-memory list has been identified, appending each pointer takes constant time. Note that the dictionary data structure is assumed to be reinitialized after each merging event.

An important consideration in any database system is the number of disk accesses required to process data. Our analysis does not directly account for the disk activity. However, we note that all techniques modeled in this article allow sequential processing of pointers, in which case the number of disk operations required to complete an operation can be derived from any analysis that is performed to count more elementary operations, such as comparisons or data movements. The only exception to this numeric relationship is the INPLACE strategy, which is used in Section 4.2 as an additional reference point.

3.1 Offline Index Construction

An obvious point of comparison for online construction algorithms is to examine the efficiency of offline algorithms whose performance forms a lower bound to that achievable by online techniques. The method of Heinz and Zobel [2003] is used throughout this article as being exemplary in the performance of offline construction algorithms, and is presented as Algorithm 1. We refer to this

Algorithm 1 [OFFLINE] Index construction using single-pass in-memory inversion

Input: a sequence of $\langle p_0, \ldots, p_{n-1} \rangle$ pointers, and b, the pointer capacity of main memory.
1: assign $T \leftarrow \{\}$ {initialize an on-disk temporary space}
2: assign $r \leftarrow 0$, $i \leftarrow 0$
3: **while** $i < n$ **do** {index further pointers}
4: assign $B \leftarrow \{\}$ {initialize a new bufferload of pointers}
{invert a bufferload of pointers}
5: **while** $|B| < b$ **and** $i < n$ **do** {further pointers fit into main memory}
6: **if** $t \notin \text{vocab}(B)$ **then** {term t does not have an associated in-memory list}
7: assign $B_t \leftarrow \langle\rangle$ {create an empty in-memory list for term t}
8: assign $B_t \leftarrow B_t + p_i$ {append pointer p_i to in-memory list B_t for term t}
9: $i \leftarrow i + 1$
{write a bufferload of pointers to disk as a sorted run}
10: assign $T_r \leftarrow B$ {store bufferload of pointers in temporary disk space, sorted by term}
11: assign $r \leftarrow r + 1$
12: merge r runs in T to form the final index D
Output: an inverted index, D, over the n pointers

algorithm as OFFLINE. A similar algorithm is given by Moffat and Bell [1995], with the key difference being the way that the vocabulary is managed.

The OFFLINE algorithm operates by building a bufferload of pointers in main memory, and writing them to disk as sorted runs once main memory is full. After all pointers have been processed in this fashion, the sorted runs are merged into the final index. Heinz and Zobel [2003] suggest using multiway *in-situ* merging [Moffat and Bell 1995] to form the final index, but the implementation of OFFLINE used in the experiments described in Section 6 employs a faster standard multiway merge using additional disk space [Knuth 1973, p. 252].

There are three distinct activities performed by all construction algorithms. The first is in-memory inversion, where a bufferload of pointers is built in main memory, as described by Heinz and Zobel [2003], resulting in a set of pointers indexed by the distinct terms to which they refer. The cost of this activity is denoted by α, and is the same for all of the algorithms considered in this article. The other two activities are the manner in which in-memory pointers are incorporated into the on-disk index during merging events, denoted β, and the manner in which the vocabulary is updated to reflect such changes, denoted by γ.

The cost of inverting pointers in main memory, as performed by the OFFLINE algorithm, can be reasonably approximated as $n \log b$, since $\lceil n/b \rceil$ bufferloads of pointers are inverted, at a cost of $b \log b$ each. The cost of reading the pointers must also be added, making the total cost of in-memory inversion

$$\alpha(n, b) = n + n \log b. \tag{1}$$

The OFFLINE algorithm creates inverted lists by writing sorted runs to disk and subsequently performing a multiway merge. The cost attributed to the merge requires some care to correctly account for the comparisons performed. A simple

approach is to consider that the merge reads a total of n pointers from a set of $\lceil n/b \rceil$ sorted runs, requiring a total of $n \log_2 \lceil n/b \rceil$ comparisons. But this is pessimistic, because the pointers created during in-memory inversion are already ordered by document identifier, and the bufferloads index disjoint portions of the collection. Once a pointer is selected, all further pointers in that bufferload for that term can also be transferred to the final index.

Taking this observation into account, $\sum_{i=1}^{\lceil n/b \rceil} d_i$ "find smallest pointer" operations are required during the merge, where d_i is the number of distinct terms in the ith bufferload. In the worst case, that sum may become n if terms occur only once in each bufferload, but this is unrepresentative of typical text collections. An alternative might be to assume a linear relationship between distinct terms and pointers [Williams and Zobel 2005]. However, linear estimates become increasingly pessimistic for large sets of pointers of the kind considered here.

3.2 Vocabulary Size

We proceed by assuming that the number of distinct terms arising from a sequence of x pointers is bounded by $x/\log_2 x$; that is, that the average multiplicity of the words appearing in a sequence of x words is $\log_2 x$. This is a slightly stronger assumption than linearity, in that, in the limit, $x/\log_2 x$ is smaller than x/k for all values of k. Nevertheless, it provides a generous empirical bound (demonstrated in Section 6.2), and has the desired behavior when subsequences of pointers are considered. For example, we are suggesting that a sequence of 100 words would give rise to around 15 distinct words; and a sequence of one million words would contain around 50,000 distinct words. The assumption of vocabulary growth rates has similarities to Heap's law [Baeza-Yates and Ribeiro-Neto 1999, p. 147], which states that the number of distinct terms, v, is bounded above by $O(n^\phi)$, where the growth parameter ϕ depends on the given collection. Our assumption of $v \le n/\log n$ avoids parameters in favor of a looser upper bound that is sufficient for our needs here.

On this basis, we have $d_i \le b/\log b$, and the cost of inverted list construction using the OFFLINE algorithm can be quantified:

$$\begin{aligned} \beta_{\text{OFFLINE}}(n, b) &= n + \log\lceil n/b \rceil \times \sum_{i=1}^{\lceil n/b \rceil} d_i \\ &\approx n + \frac{n \log(n/b)}{\log b}\,. \end{aligned} \tag{2}$$

If b is taken to be constant, the practice that we adopt throughout this article, then the cost of offline index construction is asymptotically dominated by the $O(n \log n)$ cost of the multiway merge. Worth noting also is that if b is at least $\sqrt{n}$ (which is the case in all of our experiments), then $\beta_{\text{OFFLINE}}(n, b) = n + (n \log(n/b))/\log b \le 2n$, and the $n \log b \approx (n \log n)/2$ in-memory inversion cost dominates.

In addition to the costs associated with building inverted lists, the OFFLINE algorithm must also construct a vocabulary. This can be achieved using a B$^+$-tree bulk-loading algorithm [Ramakrishnan and Gehrke 2003, p. 360], where

each entry is successively written into leaf pages. After any page has been filled, the first entry in that page is written into the index page that is its hierarchical parent. We assume that an average of $f > 1$ entries fit into each page of the vocabulary, with a typical value of perhaps $f \approx 200$. The one-time vocabulary construction cost is then

$$\gamma_{\text{V-OFFLINE}}(n) = \sum_{i=0}^{\infty} \left\lceil \frac{n}{f^i \log n} \right\rceil \approx \frac{n}{\log n} \times \sum_{i=0}^{\infty} \frac{1}{f^i}.$$

This quantity can be reduced further, given $f \gg 1$, by noting that

$$\sum_{i=0}^{\infty} \frac{1}{f^i} = \frac{f}{f-1} \approx 1.$$

Therefore

$$\gamma_{\text{V-OFFLINE}}(n) \approx \frac{n}{\log n}. \tag{3}$$

Note that to prevent confusion with list construction algorithms, we add a leading "V-" to vocabulary update processes. The reasons for this distinction will become clearer in Section 5.4.

Having modeled the cost of each of the three components of OFFLINE construction, we can reason about the overall running time of the OFFLINE algorithm by considering how the three costs might be combined. The $O(n/\log n)$ cost of vocabulary construction is, asymptotically, the least costly of the three processes. The cost of the multiway merge, though $O(n \log n)$ for any fixed value of b, is (because b is typically large relative to $\sqrt{n}$) near-linear in practice, meaning that the $O(n \log b)$ cost of in-memory inversion dominates. Replacing the comparison-based dictionary structure assumed in the analysis here with one based on hashing allows $O(1)$-time per pointer to be achieved, but at the end of each bufferload the vocabulary must then be sorted, at a total cost of $O(n \log b)$ time. Either way, the constant factor is small, and for large-but-fixed values of b, experiments show that the time required for offline index construction varies approximately linearly with index size.

To quantify some of the calculations, typical values for the two parameters are $n \approx 2 \times 10^{10}$ and $b \approx 8 \times 10^7$. An index of 20 billion pointers might arise, for example, in a collection of 500 GB of Web documents; and 80 million pointers stored in the in-memory part of the index would require approximately 500 MB of main memory, and give rise to about 200 MB of compressed pointers on disk. Using these figures, the total cost of OFFLINE index construction amounts to just 30 billion operations above the 550 billion operations required as the base cost of the in-memory inversion of the $\lceil n/b \rceil$ bufferloads of pointers.

4. DYNAMIC INDEXES

Three online index construction approaches have been explored in previous literature [Lester et al. 2005]. They are to rebuild the index from the stored collection (termed *rebuild*), to merge the bufferload of new pointers into the

index (termed *remerge*), or to perform an in-place update of each inverted list that has new pointers (termed *inplace*).

Rebuilding is a relatively wasteful strategy for online index construction. Instead of altering the existing index in response to collection modifications, previous pointers are discarded and the collection is reprocessed. In contrast, in both the inplace and remerge approaches the previous index is extended to include new pointers. During merging events, the inplace and remerge strategies can either maintain each inverted list in a single contiguous disk location, or to allow it to occupy multiple, discontiguous locations. Indexes constructed offline typically store each inverted list in single contiguous extent, as this provides maximum query performance. Breaking lists into discontiguous fragments increases online indexing speed, at the cost of requiring additional disk operations during query processing.

4.1 Rebuilding

The simplest online index construction 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 strategy, it is not unreasonable for small collections with low update frequency. For example, Lester et al. [2005] give experimental results showing that this technique is plausible in some restricted situations.

At each merging event, the REBUILD strategy constructs a new index from the stored bufferloads of data processed thus far, using the OFFLINE construction algorithm. Once the new index is ready, the old is discarded, and queries can be resolved using the updated index. At any point during the execution of the algorithm, a query can be resolved using the combination of the on-disk and in-memory index components stored by REBUILD.

During each merging event, REBUILD reprocesses the previous bufferloads against the new information provided as part of the most recent bufferload. Because the growth of pointers is linear, the cost of maintaining the inverted lists over an eventual set of n pointers is thus $n/(2b)$ times the cost of the OFFLINE method:

$$\begin{aligned}
\beta_{\text{REBUILD}}(n,b) &= \sum_{i=1}^{\lceil n/b \rceil} (\alpha(ib,b) + \beta_{\text{OFFLINE}}(ib,b)) \\
&\approx \sum_{i=1}^{\lceil n/b \rceil} \left(ib + \frac{ib \log \lceil ib/b \rceil}{\log b} \right) \\
&\leq b(1+\log_b(n/b)) \sum_{i=1}^{\lceil n/b \rceil} i \\
&\approx \frac{n^2}{2b}(1+\log_b(n/b)), \qquad (4)
\end{aligned}$$

where it is assumed that the source text does not need to be completely reparsed after each bufferload and that all intermediate bufferloads can be retained on disk.

An upper bound on the cost of maintaining the vocabulary using REBUILD can be similarly determined by observing that the vocabulary is processed n/b times, with at most $n/\log n$ entries present at each iteration:

$$\gamma_{\text{V-REBUILD}}(n, b) \approx \frac{n^2}{b \log n}. \tag{5}$$

That is, for any fixed value of b, REBUILD requires time that grows (approximately) quadratically to maintain the inverted lists, with $O(n^2/\log n)$ time required to maintain the vocabulary. For the values of n and b hypothesized above, the β cost amounts to some 3.3 trillion operations, and the γ cost amounts to around 150 billion operations. Because of the repeated computations, the overall construction cost of 4.0 trillion operations is nearly a full order of magnitude greater than the comparable cost of OFFLINE construction.

Note that we are not arguing that this method is efficient, and it is discussed purely because it is one poss*ib*le way of supporting the requirement for online index construction.

4.2 In-place Update

The second merging strategy involves writing new pointers at the end of the existing lists whenever possible. 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 moved to occupy unused space in a new location in the file. Variations include keeping short 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 lists to reduce relocations and discontiguity [Tomasic et al. 1994; Shieh and Chung 2005]. The staggered growth of inverted lists also introduces a free-space management problem. Space management of large binary objects, such as image data, has been examined by several authors [Biliris 1992a, 1992b; Carey et al. 1986, 1989; Lehman and Lindsay 1989], but note that inverted indexes present different problems to those of other types of binary data.

Algorithm 2 specifies the operation of INPLACE construction strategies. After a bufferload of pointers has been inverted, each term with in-memory pointers is visited in turn, and the on-disk portion of that inverted list is located. If the term is new, the in-memory pointers for that term are written to disk in any location where there is sufficient free space. If the term has previous pointers, then the amount of free space following that list is ascertained and the new pointers appended if possible. Lists with insufficient free space are relocated to a new position on disk, by finding a suitable new location, reading the previous list from its current location, and writing the entire updated list to the new one. Whenever a list is moved to a new location on disk, over-allocation factors $k \geq 1.0$ and $a \geq 0$ are used to reserve additional free space at the end of the list. Two variations of Algorithm 2 are of particular interest: INPLACE, with geometric over-allocation only (that is, $k \geq 1.0$ and $a = 0$) termed INPLACE GEOM; and INPLACE, with arithmetic over-allocation only ($k = 1.0$ and $a > 0$) termed INPLACE ARITH.

Algorithm 2. [INPLACE] On-line index construction using individual update of lists and over-allocation.

Input: a sequence of $\langle p_0, p_1, \ldots \rangle$ pointers, b, the pointer capacity of main memory, and over-allocation constants $k \geq 1.0$ and $a \geq 0$.

```
 1: assign D ← {} {initialize the on-disk index}
 2: assign i ← 0
 3: while true do {index further pointers}
 4:    assign B ← {} {initialize a new bufferload of pointers}
       {invert a bufferload of pointers}
 5:    while |B| < b do {further pointers fit into main memory}
 6:       if t ∉ vocab(B) then {term t does not have an associated in-memory list}
 7:          assign B_t ← ⟨⟩ {create an empty in-memory list for term t}
 8:       assign B_t ← B_t + p_i {add pointer p_i to in-memory list B_t for term t}
 9:       assign i ← i + 1
       {incorporate the bufferload of pointers into the on-disk index}
10:    for each index term t ∈ vocab(B) do {for all terms t that have an in-memory list}
11:       if t ∈ vocab(D) then {term t also has on-disk pointers}
12:          determine the amount of available space w available within and
             following D_t
13:          if w < |D_t| + |B_t| then {previous list D_t must be relocated}
14:             copy list D_t to a new location with available space not less than
                k × (|D_t| + |B_t|) + a
15:          append B_t to D_t {add new pointers to previous list}
16:       else {term t is new to D}
17:          find a location with available space not less than k × |B_t| + a
18:          assign D_t ← B_t {write in-memory pointers into a new on-disk list}
```

Invariant: The combination of D and B indexes all pointers examined thus far

To analyze the cost of building an index using the INPLACE GEOM approach, suppose that each list is over-allocated by a fixed proportional factor of $k > 1$. When the list for some term t contains f_t pointers and is at its current capacity, and a $f_t + 1$st pointer is to be added to it during the merge event, the list is first extended to be $\lceil k f_t \rceil$ pointers and then the new one is 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 $f_t - f_t/k$ items have just been copied for the first time; $f_t/k - f_t/k^2$ 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

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

For example, if $k = 1.25$, then just after any 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 highest. The expression above is thus an upper bound on the per-pointer cost of constructing the inverted lists.

The cost of adding every pointer to its initial list should also be included. Taking this into account, the total number of operations required to process the n pointers given by

$$\beta_{\text{INPLACE GEOM}}(n) = \frac{(2k-1)}{k-1} n\,. \tag{6}$$

No similar guarantee applies to the INPLACE ARITH algorithm, where—assuming that the over-allocation factor a is smaller than b, the size of the in-memory buffer—every pointer in the index may have to be moved for each successive merging event. As a result, the worst-case cost of inverted list construction using the INPLACE ARITH algorithm is

$$\begin{aligned} \beta_{\text{INPLACE ARITH}}(n, b) &\le \sum_{i=1}^{\lceil n/b \rceil} ib \\ &\approx \frac{n^2}{2b}\,. \end{aligned} \tag{7}$$

For any fixed value of b, the complexity of the INPLACE ARITH algorithm is asymptotically dominated by the $O(n^2)$ cost of pointer relocation. Thus the INPLACE ARITH algorithm scales poorly to large collections compared to an in-place approach with geometric over-allocation. For example, when n and b are as suggested at the beginning of this section, and $k = 1.25$ is used, the INPLACE GEOM and INPLACE ARITH algorithms require, respectively, around 120 billion and 2.5 trillion operations.

However, the INPLACE GEOM method also has two disadvantages that partly or completely negate the small operation count. The first is its memory overhead. Because all the lists are growing, around 60% of the over-allocated space is always vacant. That is, when $k = 1.25$, around 15% of the disk space allocated to the lists in the index is unused. There will also be external fragmentation that is not taken into account in this computation, caused by the unpredictable overall sequence of list resizings. This could easily add a further 5% to 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. Data movements are certainly a critical cost, but the inplace mechanisms also require nonsequential processing of the index file during each merge event, and it is the consequent disk accesses that make it expensive in practice [Lester et al. 2005].

Vocabulary entries for altered inverted lists are individually updated in these strategies. In terms of comparisons performed, the cost of each vocabulary update is not more than $\log(n/\log n) \le \log n$. Over all (n/b) bufferloads, and the $d_i = (b/\log b)$ distinct terms in each bufferload, the total vocabulary cost is thus

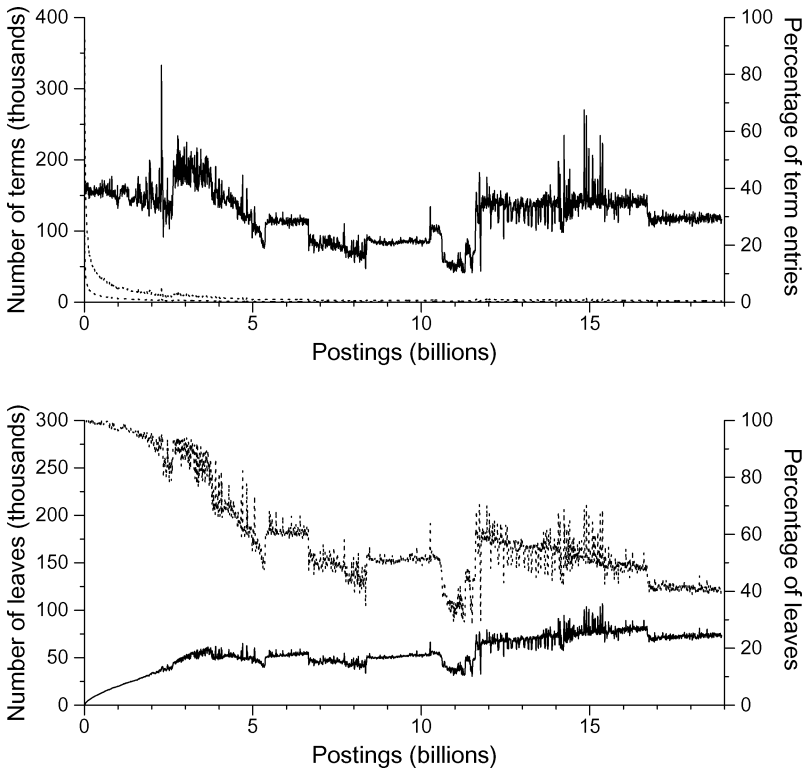

Fig. 1. The number (solid line) and percentage (dotted line) of vocabulary entries (upper graph) and vocabulary pages (lower graph) that are accessed during each merging event. The horizontal axes show the number of pointers in the index. The left vertical axes show the number of items requiring change per merging event in thousands; and the right vertical axes show the same value as a percentage of the total number of vocabulary items or pages. The collection used is the 426 GB gov2 [Clarke et al. 2004], with a buffer of $b = 8$ million pointers and a vocabulary page size of 4 kB.

bounded by

$$\gamma_{\text{V-INPLACE}}(n, b) = \frac{n}{b} \frac{b}{\log b} \log \frac{n}{\log n} \approx n \log_b n \,. \tag{8}$$

As was the case above, if b is at least $\sqrt{n}$, this expression is linear in n.

A possible issue with the vocabulary analysis is that each comparison might correspond to a disk access to fetch a page of the B$^+$-tree, meaning that the linearity might have a high constant factor. The top graph in Figure 1 quantifies this relationship, by plotting the number of vocabulary entries that require alteration during each merging event when indexing the 426 GB gov2 collection [Clarke et al. 2004] with a (small) buffer size of $b = 8 \times 10^6$. The number of vocabulary entries requiring alteration is the d_i value for that bufferload, and is plotted as a solid line. Note that for $b = 8 \times 10^6$, the expression $b/\log b$ has the value 350,000, and is not exceeded in any of the bufferloads in this text sample. Note also that to avoid anomalies resulting from the partial bufferload produced at the end of the index construction, the final point is not included. The percentage of total vocabulary entries requiring alteration is plotted as the

dashed line. As predicted, the number of entries updated in each merging event is a decreasing fraction of the total vocabulary size as the vocabulary grows.

However, the vocabulary resides within an on-disk B$^+$-tree, and we expect disk transfer times for page accesses to be a significant cost. As a close surrogate for the total number of pages changed by the vocabulary update strategy, the number of leaf pages altered by the V-REMERGE strategy is shown in the bottom graph of Figure 1. The number of B$^+$-tree leaves changed has a strong correlation to the number of terms with new pointers in each merging event. However, the blocking of vocabulary entries into B$^+$-tree leaves ensures that a far higher proportion of leaf pages is altered than of vocabulary entries, and nearly every merging event affects 40% or more of the leaf pages. We return to this observation below.

4.3 Remerge Update

The third index update strategy avoids random accesses by sequentially reprocessing the entire index at each merging event. To process a bufferload of pointers, the entire on-disk index is read 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 is transferred over to the new index. This approach is shown by Algorithm 3, which defines the REMERGE strategy. It 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, peak disk usage will also be twice the cost of storing the index [Clarke and Cormack 1995; Moffat and Bell 1995]. However, compared to the inplace methods, the remerge approach has the advantage of performing all disk operations sequentially.

The merge between the in-memory pointers and the on-disk pointers, both of which are sorted, incurs a data transfer cost proportional to the total number of pointers in the resulting index. Over the n/b merging events the cost is thus

$$\beta_{\text{REMERGE}}(n, b) = \sum_{i=1}^{\lceil n/b \rceil} ib \approx \frac{n^2}{2b}. \tag{9}$$

The number of comparisons required by the merge is not represented in Eq. (9), as there are exactly two sources in each merge. The resulting cost of merge comparisons is dominated by the $O(n^2)$ cost of processing pointers. This cost is identical to the limit derived for the INPLACE ARITH algorithm in Eq. (7), reflecting the fact that both algorithms do (in the case of REMERGE) or may (in the case of INPLACE ARITH) reprocess all pointers within the index during each merging event.

Construction of a vocabulary during REMERGE is unchanged from the process required during REBUILD—that is, bulk construction of a b+-tree during inverted list merging. Analysis of the number of operations utilized by V-REMERGE is thus

Algorithm 3. [REMERGE] On-line index construction using repeated merging.

Input: a sequence of $\langle p_0, p_1, ... \rangle$ pointers, and b, the pointer capacity of main memory.

```
1:  assign D ← {} {initialize the on-disk index}
2:  assign i ← 0
3:  while true do {index further pointers}
4:    assign B ← {} {initialize a new bufferload of pointers}
    {invert a bufferload of pointers}
5:    while |B| < b do {further pointers fit into main memory}
6:      if t ∉ vocab(B) then {term t does not have an associated in-memory list}
7:        assign B_t ← ⟨⟩ {create an empty in-memory list for term t}
8:      assign B_t ← B_t + p_i {append pointer p_i to in-memory list B_t for term t}
9:      assign i ← i + 1
    {incorporate the bufferload of pointers into the on-disk index}
10: assign T ← {} {initialize a new on-disk index}
11: for each term t ∈ vocab(D) ∪ vocab(B), in lexicographical order do
12:   if t ∈ vocab(D) then {term t has on-disk pointers}
13:     assign T_t ← D_t {append on-disk pointers D_t for term t to the new index}
14:   if t ∈ vocab(B) then {term t has in-memory pointers}
15:     if t ∈ vocab(T) then
16:       assign T_t ← T_t + B_t {append in-memory pointers to existing list in new
            index}
17:     else
18:       assign T_t ← B_t {merge the new list into the index}
19: assign D ← T {replace previous index}
```

Invariant: The combination of D and B indexes all pointers examined thus far

identical to the analysis of V-REBUILD:

$$\gamma_{\text{V-REMERGE}}(n, b) \approx \frac{n^2}{b \log n}, \tag{10}$$

that is, a maximum of $n/\log n$ entries are processed n/b times. With a buffer of constant size b, this vocabulary update cost is not quite quadratic, suggesting that the $O(n^2)$ cost of inverted list construction should be the dominant factor in the performance of REMERGE implementations.

The number of operations required by REMERGE in the example given earlier, with $n \approx 2 \times 10^{10}$ and $b \approx 8 \times 10^7$, is 2.5 trillion operations for inverted list construction, and 146 billion operations to maintain the vocabulary, making it (based purely on the operation count) as costly as REBUILD and INPLACE ARITH. However, in experiments with these three methods, Lester et al. [2005] show that REMERGE is more efficient than the other two in a wide range of practical scenarios, primarily because of its sequential disk access pattern.

4.4 Multiple Partitions

One of the primary reasons that these three online index construction algorithms are inefficient is that they keep a single, contiguous, inverted list

per term. On the other hand, sort-based offline construction algorithms are not constrained in this regard, and make use of multiway merging strategies to reduce the number of times each pointer is handled. For example, the dominant operation in the OFFLINE algorithm is the in-memory inversion of bufferloads of pointers, the cost of which on a per-pointer basis is independent of the total amount of data being indexed. Tomasic et al. [1994] describe an index construction strategy that avoids reprocessing 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 pointers. Tomasic et al. do not use over-allocation for any but the first fragment, so their algorithm creates approximately one fragment per merging 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

$$\beta_{\text{TOMASIC}}(n) \approx n \tag{11}$$

plus the time needed to create the chains of pointers that thread the structure together. Query costs scale badly in this approach, or with similar approaches in which lists are 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 250 disk seeks per query term.

5. GEOMETRIC PARTITIONING

We now describe a new scheme that blends the remerge method described in Section 4.3, and the too-many-fragments approach of Tomasic et al. [1994]. 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. The drawback is that each index list is now in multiple parts, making querying slower than with single-partition lists. Section 6 quantifies the amount of that degradation.

5.1 Hierarchical Merging

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 to enter the collection are indexed via the largest partition. Figure 2 shows an example arrangement in which the on-disk part of the index is split into three partitions.

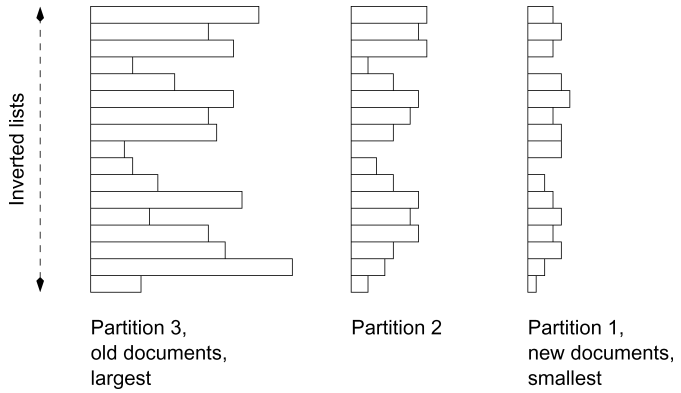

Fig. 2. A geometrically partitioned index structure, with three levels. The oldest index pointers are in the largest partition, which in this example is at level 3. The vocabulary, not shown in the figure, includes three pointers with each term's entry. Some terms are absent from some partitions.

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 work of Tomasic et al. The difference in our approach is that we ensure, via periodic merging, that the number of partitions does not grow excessively.

Consider what happens when an in-memory bufferload of pointers 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 merging 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 jth partial index not more than $(r-1)r^{j-1}b$ pointers. In addition, r also specifies a lower bound on the size of each partition—at level j the partition is either empty, or contains at least $r^{j-1}b$ 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 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 $r = 3$,

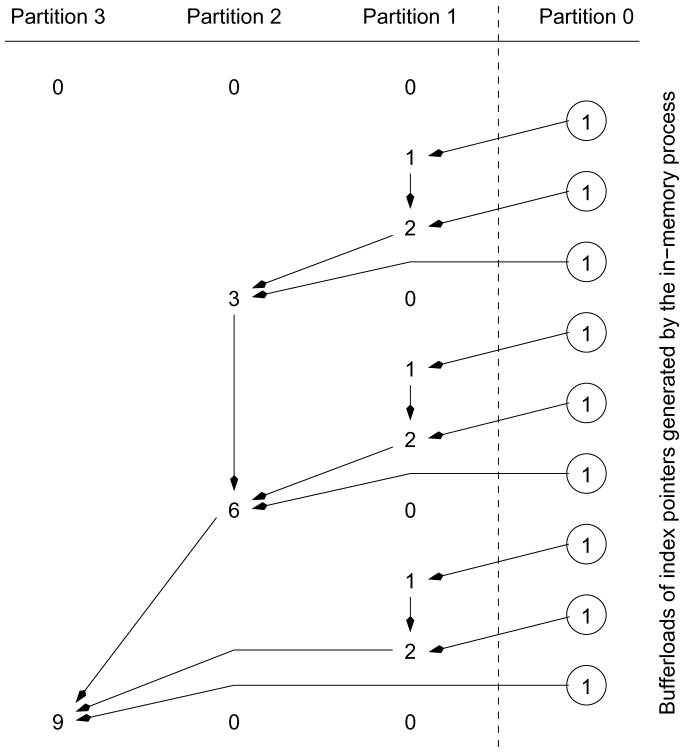

Fig. 3. The merging pattern established when $r = 3$. The first index is placed into partition 3 only after nine bufferloads have been generated by the in-memory part of the indexing process. All numbers listed represent multiples of b, the size of each bufferload.

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 $3b$ pointers that are the result of this merge are 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 3 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.

5.2 Analysis

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

one full cycle this sequence sums to

$$\frac{(r-1)r}{2} r^{j-1} = \frac{(r-1)r^j}{2}. \tag{12}$$

We make use of this quantity below.

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 3, 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 an index of n pointers in total, the merging pattern has $\lceil n/b \rceil$ rows. Hence, in the ith column there will be at most $\lceil n/b \rceil / r^i$ full cycles of the merging pattern; furthermore, the average merging 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 given by

$$p = 1 + \lceil \log_r(n/b) \rceil \approx 0.5 + \frac{\log(n/b)}{\log r}. \tag{13}$$

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

$$\begin{aligned} \beta_{\text{GEOM}}(n, b, r) &= b \times \sum_{i=1}^{p} \frac{\lceil n/b \rceil}{r^i} \frac{(r-1)r^i}{2} \\ &\approx \frac{(r-1)n}{2} \left(0.5 + \frac{\log(n/b)}{\log r} \right). \end{aligned} \tag{14}$$

Breaking each inverted list into partitions introduces additional disk seek operations, and slows overall query throughput rates, since index list components for each term need to be fetched from all nonzero partitions. 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

$$\frac{r-1}{r} \left(0.5 + \frac{\log(n/b)}{\log r} \right). \tag{15}$$

That is, Eq. (15) quantifies the number of disk accesses that are required to retrieve the full inverted list of each term. Table I shows, for several values of r, the number of operations required during merging events to build the index (in billions of operations, calculated according to Eq. (14)), and the expected number of disk accesses per term (using Eq. (15)), for the values $n = 10^{10}$ and $b = 8 \times 10^7$. 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.

However, the cost attributed to each multiway merge in the analysis requires elaboration. In Figure 3, for example, it was assumed that the three-way merge to create the partitions of size six and nine cost six and nine units

Table I.

r	$\beta_{\text{GEOM}}(n, b, r)$	Access Cost
2	85	4.2
3	110	3.7
4	130	3.4
6	180	3.0
8	220	2.8
12	300	2.5

Index construction cost (billions of operations calculated) and the expected number of disk accesses per query term (calculated), for $n = 2 \times 10^{10}$ and $b = 8 \times 10^7$, and different fixed values of r.

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 $\sum_{i=1}^{k} n_k$. To demonstrate that the linear summation is correct, first consider the total number of source partitions involved in merges throughout the construction of the index, as shown by Figure 3. Partition 0 participates in all of the $\lceil n/b \rceil$ merges as a source. Every other partition i, where $i > 0$, is used as a source for $r-1$ out of every r^i merges. Thus, the total number of source partitions, s, merged during the construction of the index is given by

$$\begin{aligned} s &\approx \left\lceil \frac{n}{b} \right\rceil + \sum_{i=1}^{p} \frac{r-1}{r^i} \left\lceil \frac{n}{b} \right\rceil \\ &\leq \left\lceil \frac{n}{b} \right\rceil \left(1 + (r-1) \times \sum_{i=1}^{\infty} \frac{1}{r^i} \right). \end{aligned} \tag{16}$$

Given that $r > 1$, this expression can be simplified using the identity $\sum_{i=1}^{\infty}(1/r^i) = 1/(r-1)$. Averaging the total number of source partitions s, over the number of merges $\lceil n/b \rceil$, shows that on average just two partitions participate in each merge. Finally, since the log function is concave, the cost of comparison-based merging must also be linear in the number of data items, even without making the earlier assumption that the number of distinct terms appearing in a sequence of x pointers is at most $x/\log x$.

5.3 Varying the Radix

The best choice of r depends on the balance of operations. Table I 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.

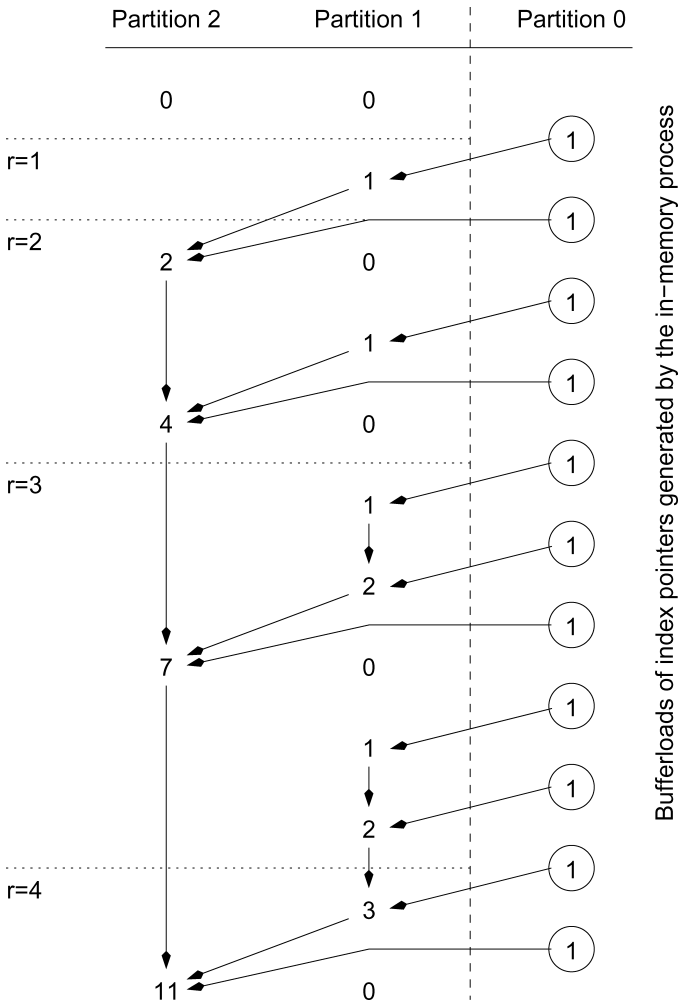

Fig. 4. The merging pattern established when $p = 2$ and r is varied. All numbers listed represent multiples of b, the size of each bufferload.

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

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

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 4 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

Table II.

p	Final r	$\beta_{\text{GEOM}}(n, b, (n/b)^{1/p})$
2	16	320
3	7	190
4	4	160
5	4	150

Inverted list construction cost (billions of operations, calculated) for $n = 2 \times 10^{10}$ and $b = 8 \times 10^7$, and different fixed values of p. The final value of r in each case is also shown.

(because r is increased to 5), 25, 31, and then 38. The REMERGE strategy of Section 4.3 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

$$\begin{aligned}\beta_{\text{GEOM}}(n, b, r) &= \beta_{\text{GEOM}}(n, b, (n/b)^{1/p}) \\ &= n \times \frac{(n/b)^{1/p} - 1}{2} \times (0.5 + p) \\ &\approx \frac{p n^{1+1/p}}{2 b^{1/p}}, \end{aligned} \tag{17}$$

which is asymptotically dominated by the $O(n^{1+1/p})$ term. Table II shows the calculated number of operations required to build an index for the hypothesized values of n and b, for various values of the bound p. Table III then shows the calculated cost of the bufferload-processing part of all of the index construction mechanisms we have described. To set these values in context, the in-memory inversion cost (α) under the two scenarios presented in Table III are 480 billion operations for $b = 8 \times 10^6$ and 550 billion operations for $b = 8 \times 10^7$, supposing throughout that $n = 2 \times 10^{10}$.

5.4 Vocabulary Update

A subtle issue that arises with geometric partitioning of inverted lists is how the vocabulary should be updated. Since the inverted list partitions are updated by merging, a matching V-REMERGE strategy is a reasonable approach in combination with GEOM strategies. However, as indicated by Eq. (10), the quadratic complexity of the REMERGE approach extends to its use as a vocabulary update scheme, and experiments in Section 6 demonstrate that for large collections, or small buffer sizes, the cost of the V-REMERGE strategy does indeed dominate the cost of geometric list merging. The issues presented in Figure 1 suggest that use of a V-INPLACE strategy with geometric partitioning—that is, individual update of each vocabulary entry that changes during merging events—is also unlikely to be an improvement over V-REMERGE.

To address this dilemma, geometric partitioning techniques need to be applied to vocabulary update as well as inverted list construction. The resulting V-GEOM scheme stores the vocabulary in multiple B$^+$-tree structures, merging new entries and existing vocabulary partitions to update them. To avoid confusion with geometric partitioning applied to inverted list construction, we

refer to inverted list partitions as ℓ-*partition*s and vocabulary partitions as v-*partition*s.

Vocabulary partitioning operates by storing independent vocabulary data structures. The growth of v-partitions occurs analogously to the geometric partitioning of inverted lists, and the same geometric partitioning algorithm—with a fixed radix or a fixed number of partitions—is used to update the vocabulary as is applied to the inverted lists. However, in order to further control the number of v-partitions and minimize overhead on query processing, a single v-partition can be used to record list locations within multiple ℓ-partitions. The cardinality of this relationship, $c \geq 1$, is an additional parameter required by vocabulary update using geometric partitioning. That is, there will be $\lceil p/c \rceil$ v-partitions, when geometric partitioning has created p different ℓ-partitions, and the radix for the V-GEOM algorithm is r^c compared to the r radix used by the GEOM strategy. Substituting these alternatives into the analysis developed for the GEOM algorithm (Eq. (14)) with a suitably changed radix and number of partitions, and multiplying it by the number of distinct terms in a bufferload, $b/\log b$, rather than the number of pointers in a bufferload, b, yields

$$\begin{aligned}\gamma_{\text{V-GEOM}}(n, b, r, c) &\leq \frac{b}{\log b} \times \sum_{i=1}^{\lceil p/c \rceil} \frac{\lceil n/b \rceil}{r^{ci}} \frac{(r^c - 1)r^{ci}}{2} \\ &\approx \frac{n(r^c - 1)}{2c \log b} \left(0.5 + \frac{\log(n/b)}{c \log r}\right). \end{aligned} \tag{18}$$

The fixed-partitions variation of geometric partitioning can be analyzed in the same way:

$$\gamma_{\text{V-GEOM}}(n, b, p, c) \approx \frac{p n^{1+c/p}}{2c b^{c/p} \log b}. \tag{19}$$

This analysis indicates that the cost of vocabulary update can be reduced to $O(n \log n)$ using fixed-radix partitioning, or $O(n^{1+1/p})$ using a fixed number of partitions. Evidence of the practical value of this improvement appears in the next section.

6. EXPERIMENTS

To validate our analysis, we experimented with the 426 GB `gov2` web-based document collection, and measured index construction times and querying times for a range of online and offline indexing techniques. All experiments were performed on a dual-processor Intel Pentium 4 2.8 GHz machine with hyperthreading turned on, but employing only a single processor. The experimental machine had 2 GB of RAM and was under light load, with no other significant processes running. Times presented are elapsed times, including all parsing, indexing, and list merging phases. In each experiment the construction process repeatedly inverted b (with $b = 80 \times 10^6$ for the most part) pointers in memory, corresponding to approximately 200 MB of index data. When each bufferload was full, a merging event was initiated. On the `gov2` collection, $b = 80 \times 10^6$ gave rise to 237 bufferloads of pointers needing processing. Merges were performed with a total buffer size of 100 kB, split evenly between input and output

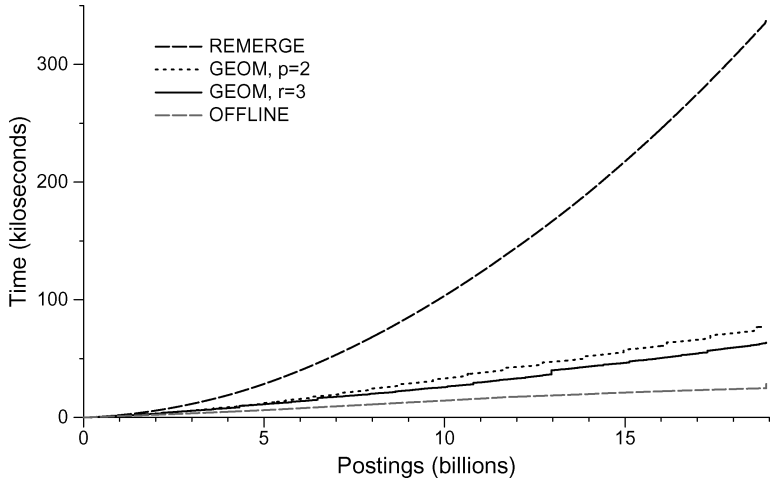

Fig. 5. Time taken to index the gov2 collection using a pointer buffer of $b = 80$ million pointers. Two variations of the GEOM algorithm are shown, one using a fixed radix of $r = 3$ and one using $p = 2$ partitions, with both using the V-REMERGE vocabulary update mechanism. In addition, the REMERGE algorithm and an OFFLINE construction baseline are shown. The horizontal axis shows the number of pointers processed. The vertical axis shows the cumulative time required to construct an index over the given number of pointers.

buffering. The half assigned for input lists was divided further between input partitions, where applicable, on an equal basis. In unreported experiments, we verified that increasing the merging buffer size did not significantly alter the outcomes of the experiments presented.

6.1 Index Construction Time

Index construction times for two different variants of the online partitioned approach are shown in Figure 5, and compared to the time taken by the OFFLINE method and the REMERGE online method. Note that the index constructed using the OFFLINE method is not available for querying until the entire collection has been processed. Times for the REBUILD and INPLACE GEOM methods are not shown; both are considerably slower than REMERGE for this combination of data and buffer size [Lester et al. 2005]. The relationships between the methods plotted in Figure 5 are as expected, with the $r = 3$ version being slightly faster than the $p = 2$ variant, and both being significantly faster than REMERGE. The super-linear growth rate of these two approaches is also as expected. In particular, note that the partitioned method with $p = 2$ forces $r = \sqrt{(n/b)}$.

Table IV explores the accuracy of the β cost models used to generate Table III. Taking the REMERGE approach as a baseline, expected percentage reductions in the β component of the predicted operation count are calculated for two versions of the geometric scheme, and compared with the computation time reductions measured when building an index for the .gov collection. The actual reductions are close to the predicted β-based reductions, validating the analysis model used in Sections 3, 4, and 5.

Table III.

		β	
Algorithm	Reference	$b = 8 \times 10^6$	$b = 8 \times 10^7$
OFFLINE	Equation 2	30	25
REBUILD	Equation 4	37,000	3,300
REMERGE	Equation 9	25,000	2,500
INPLACE GEOM, $k = 1.25$	Equation 6	120	120
INPLACE ARITH	Equation 7	25,000	2,500
GEOM, $r = 3$	Equation 14	150	110
GEOM, $p = 2$	Equation 17	1,000	320

Inverted list construction cost (billions of operations calculated) for different inverted list construction strategies, using $n = 2 \times 10^{10}$ and two different values of the pointer buffer size parameter b.

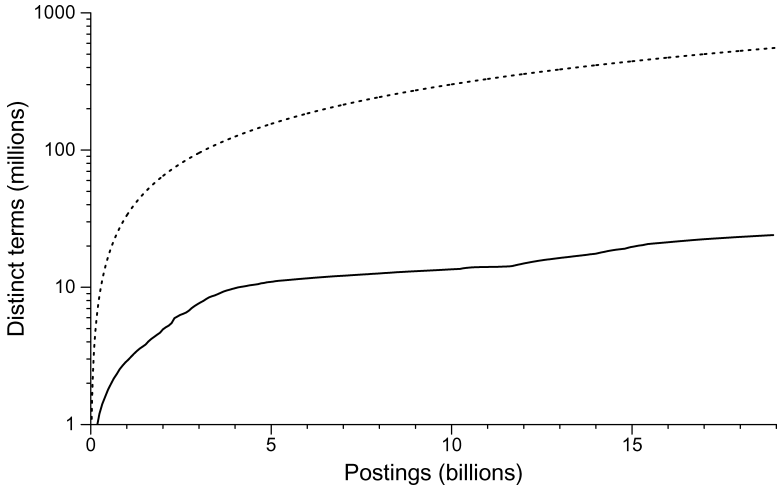

Fig. 6. The solid line shows the total number of distinct vocabulary terms during construction of an index for the gov2 collection. The assumed upper bound on the vocabulary size, $x/\log x$ for an index of x pointers, is shown by the dotted line. The vertical axis uses a logarithmic scale.

6.2 Vocabulary Size

All of the online construction methods plotted in Figure 5 utilize V-REMERGE to update the vocabulary, the cost of which grows super-linearly in the size of the collection (Eq. (10)) and is $O(n^{1.5})$, for example, if $b \approx \sqrt{n}$. That the measured performance of geometric partitioning is good indicates that the relative constant factor of vocabulary update is modest for this scenario, and that, as predicted in Section 5.2, the overall cost of construction is dominated by the cost of updating inverted lists.

Before investigating the extent to which vocabulary update costs might become important with other parameter combinations, the assumption made in Section 4 regarding the number of distinct terms produced by a sequence of pointers if length x needs to be put to the test. Figure 6 shows the relationship between the number of pointers in the index and the number of distinct terms indexed for collection gov2. It is clear that the number of distinct terms

Table IV.

Strategy	Operations (Billions)	Predicted Reduction	Build Time ('000 Seconds)	Actual Reduction
REMERGE	2,500	–	398.0	–
GEOM, $p = 2$	320	87.2%	42.2	89.4%
GEOM, $r = 3$	110	95.6%	24.3	93.9%

Predicted and actual cost reduction between REMERGE and two variants of the geometric partitioned method of index maintenance. The gov2 collection was indexed, using $b = 80$ million pointers per bufferload. The column labeled "Operations" is taken from the β costs given in Table III, with the next column, "Predicted reduction," calculating the extent of the predicted savings for the two GEOM methods relative to REMERGE, assuming that the β is the dominant component of the running time. The column labeled "Time" is then the measured cost of incremental index construction, using the V-GEOM method for vocabulary maintenance. The final column calculates actual time savings, relative to the REMERGE baseline, and provides empirical confirmation of the validity of the cost model used in Sections 3, 4, and 5.

indexed remains well within the presumed upper bound as the pointers of the collection are processed. By the time the last pointer of the collection is been dealt with, the vocabulary size is less than 10% of $n/\log n$. Further evidence of the reasonableness of the vocabulary assumption is presented in Table V, which shows the quantities $\sum_{i=1}^{\lceil n/b \rceil} d_i$ and $(\sum_{i=1}^{\lceil n/b \rceil} d_i)/(n/\log b)$ for four different collections and two different values of b. Our assumption is that each d_i value is less than $b/\log b$ (Section 3.1), meaning that the sum of the d_i values should not exceed $n/\log b$. Table V shows that this conjectured bound is met, with a factor of two safety margin, in all of our experiments.

6.3 Vocabulary Update

Figure 7 shows the time required to construct an index over the gov2 collection using geometric partitioning, with a fixed radix of $r = 3$, and the V-REMERGE strategy, and a (small) buffer of $b = 8$ million pointers. The execution time is broken into three categories: time spent appending pointers to inverted lists, β, at the top of the graph; time spent updating the vocabulary, γ, in the middle of the graph; and time spent on all other construction activities, primarily parsing and in-memory inversion, α, at the bottom. Like Figure 5, the horizontal axis shows the number of pointers incorporated into the index, and the vertical axis shows the amount of time required to construct an index over that many pointers. The numbers at the mid- and end-points of the graph are the percentage of the total time spent in each of the three cost categories, through to that point, and, as expected, the cost of in-memory inversion grows approximately linearly in the size of the collection. The cost of geometric partitioning is also modest. On the other hand, for this combination of techniques and buffer size, the cost of vocabulary update (in the middle) dominates, taking over half of the total time required for construction. In addition, the percentage of time taken by V-REMERGE at the end-point is larger than the percentage at the mid-point, confirming that the vocabulary cost is a growing fraction of the total time.

An alternative strategy is to use geometric partitioning for vocabulary update. Figure 8 shows a decomposition of the costs incurred by this strategy. Compared to Figure 7, the cost of vocabulary update has been reduced ten-fold,

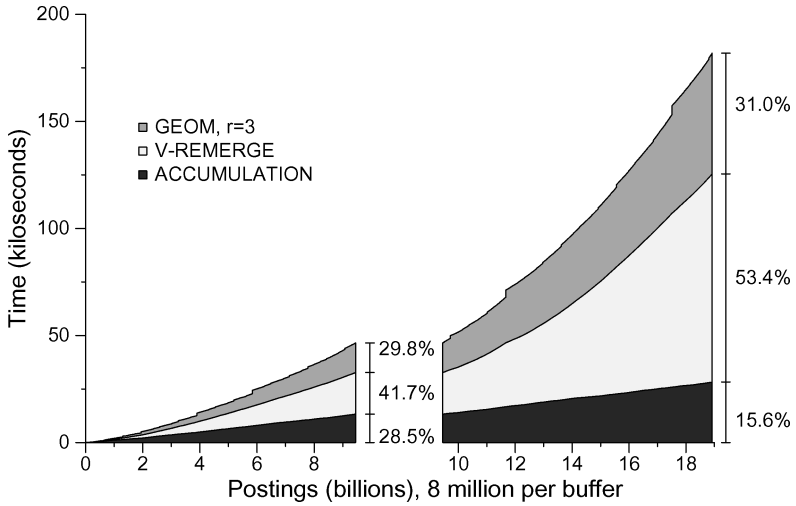

Fig. 7. Cumulative time taken for the different components of online index construction using fixed-radix geometric partitioning with $r = 3$. The vocabulary is maintained using V-REMERGE. The collection is the 426 GB GOV2, using a buffer of $b = 8$ million pointers. The cumulative time spent in each of the three cost categories is shown, with the in-memory inversion cost, α, using the darkest shading. The time spent updating the vocabulary, $\gamma_{\text{V-REMERGE}}$, is the middle component with the lightest shading, and the time spent appending pointers to inverted lists, β_{GEOM}, is shown uppermost with moderate shading. With this combination of techniques and buffer size the vocabulary costs dominate.

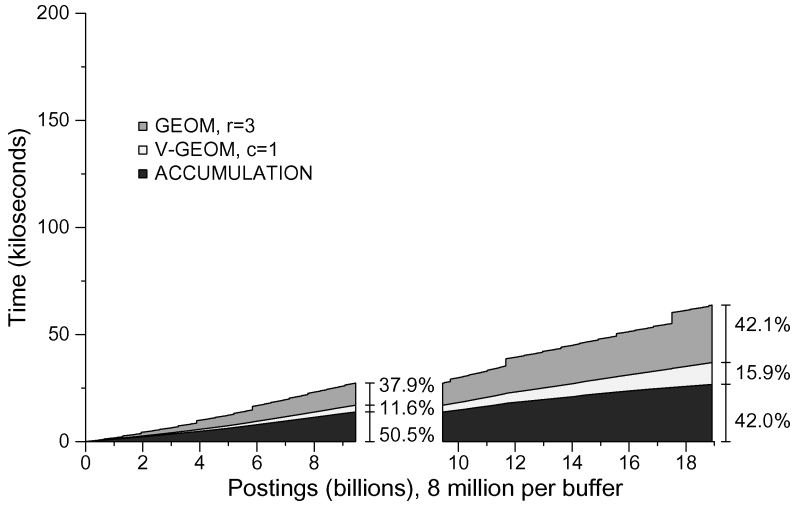

Fig. 8. Cumulative time taken for the different components of online index construction using fixed-radix geometric partitioning with $r = 3$. The vocabulary is maintained using geometric partitioning with one v-partition per ℓ-partition. All other details are as for Figure 7. The vocabulary cost no longer dominates construction time.

Table V.

			$\sum_{i=1}^{\lceil n/b \rceil} d_i$	
Collection	n	Distinct Terms	8M Pointers Buffered	80M Pointers Buffered
tipster	744.7M	1.8M	9.4M ($0.29n/\log b$)	3.8M ($0.13n/\log b$)
wt10g	902.2M	4.5M	20.1M ($0.51n/\log b$)	9.9M ($0.29n/\log b$)
gov	1.2B	3.6M	20.2M ($0.39n/\log b$)	8.7M ($0.19n/\log b$)
gov2	18.9B	24.0M	288.3M ($0.35n/\log b$)	120.3M ($0.17n/\log b$)

Collection statistics produced during construction. For each collection the total number of pointers, n, and total number of distinct terms in the collection are shown. The quantity $\sum_{i=1}^{\lceil n/b \rceil} d_i$ is recorded using $b = 8$ million pointers and $b = 80$ million pointers. The ratio between each of these values and $n/\log b$, the assumed sum of the per-bufferload vocabulary sizes, is in parentheses. The tipster [Harman 1993] collection consists of the entire contents of disks one to five of the TREC Tipster distribution. The wt10g and gov collections are described by Bailey et al. [2003] and Craswell and Hawking [2002] respectively. A suffix of "M" indicates units of millions and "B" indicates units of billions.

Table VI.

List Strategy	Vocab. Strategy	Build Time ('000 Seconds)	ℓ-Partitions	v-Partitions
REMERGE	V-REMERGE	3,100*	1	1
GEOM, $p = 2$	V-REMERGE	272.1	2	1
GEOM, $r = 3$	V-REMERGE	181.8	7	1
GEOM, $p = 2$	V-GEOM, $c = 1$	169.8	2	2
GEOM, $r = 3$	V-GEOM, $c = 4$	71.6	7	2
GEOM, $r = 3$	V-GEOM, $c = 1$	63.7	7	7
OFFLINE	OFFLINE	40.6	1	1

Time required to construct indexes for the gov2 collection, using a buffer size of $b = 8$ million pointers and different construction strategies. This buffer size produced $\lceil n/b \rceil = 2{,}364$ merging events over the collection. The maximum number of list and vocabulary partitions created by each scheme is also shown. The figure marked with an asterisk was estimated using Eq. (9) and the results of the REMERGE strategy using a larger buffer of 80 million pointers; all other figures are measured execution times on the experimental hardware. These times are higher than those presented in Table III because of the smaller buffer size b used.

making it the smallest cost category. Overall, a three-fold reduction in construction time is obtained. Table VI gives numeric details of this result, and others in the evolving sequence of techniques that we have described, all measured under the same conditions using a buffer of $b = 8$ million pointers. The strategy shown in Figure 8 is only 50% slower than OFFLINE index construction; a significant achievement, given the small buffer size used.

6.4 Query Time

The final component of our experimentation was an exploration of the effect that noncontiguous lists had on query costs. Figure 9 shows the effect of the partitioned index construction scheme on querying efficiency when $r = 3$ and $c = 1$. The upper graph shows the number of nonempty partitions at each stage in the construction process, essentially counting the number of nonzero digits in the base-r representation of the number of bufferloads processed. As geometric partitioning has been utilized for both inverted list construction and vocabulary update and $c = 1$, each nonempty partition results in both one v-partition and one ℓ-partition.

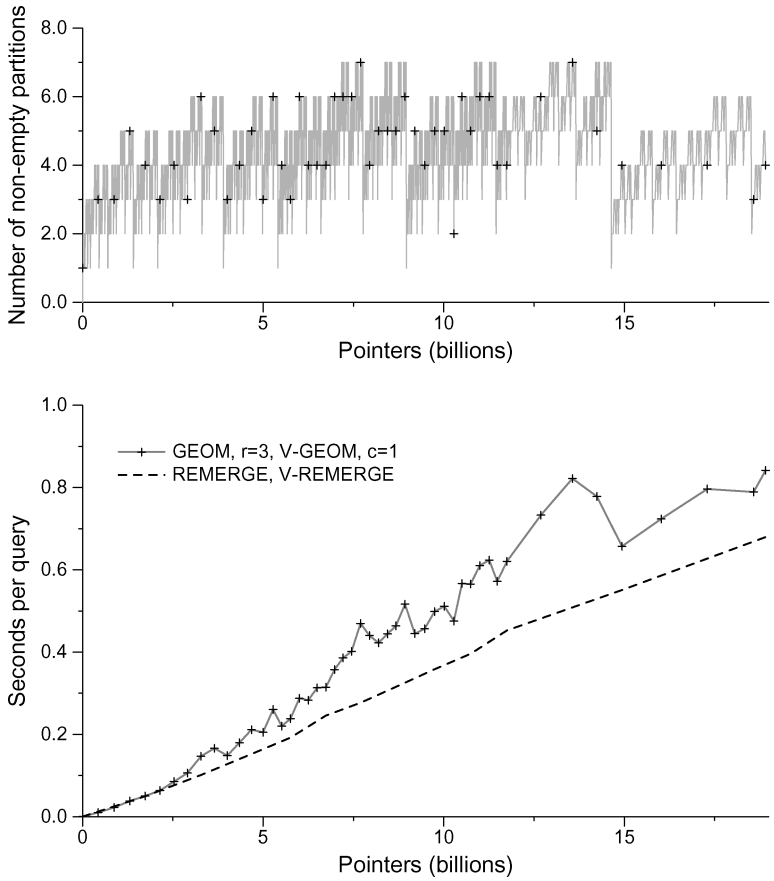

Fig. 9. Query times for a fixed-radix geometric partitioning strategy with $r = 3$, and vocabulary partitioning with one v-partition per ℓ-partition. The collection is gov2, with a buffer of approximately 8 million pointers. The upper graph shows the number of non-empty partitions after each merging event. The lower graph shows the measured per-query cost of evaluating 10,000 MSN queries, using two indexes: one with noncontiguous index lists and one with contiguous lists. In the lower graph the set of queries was executed after every approximately 50 merging events to obtain the times indicated by the tick marks.

The lower graph in Figure 9 shows the average per-query elapsed time over a sequence of 10,000 queries, using the index at 49 different points during its construction. The query sample points were dictated by the sizes of the set of files storing the collection, and are indicated by the tick marks. The 10,000 queries were taken from a query log provided by Microsoft Search, with queries selected for inclusion in the log only if they had a result from the .gov domain in the top three rank positions at the time they were executed by Microsoft Search. Preliminary experiments not described here confirmed that execution of 10,000 queries was sufficient to obtain stable measurements of execution time. In particular, the query set was large enough that data cached in main memory from previous experiments provided no improvement in running times,

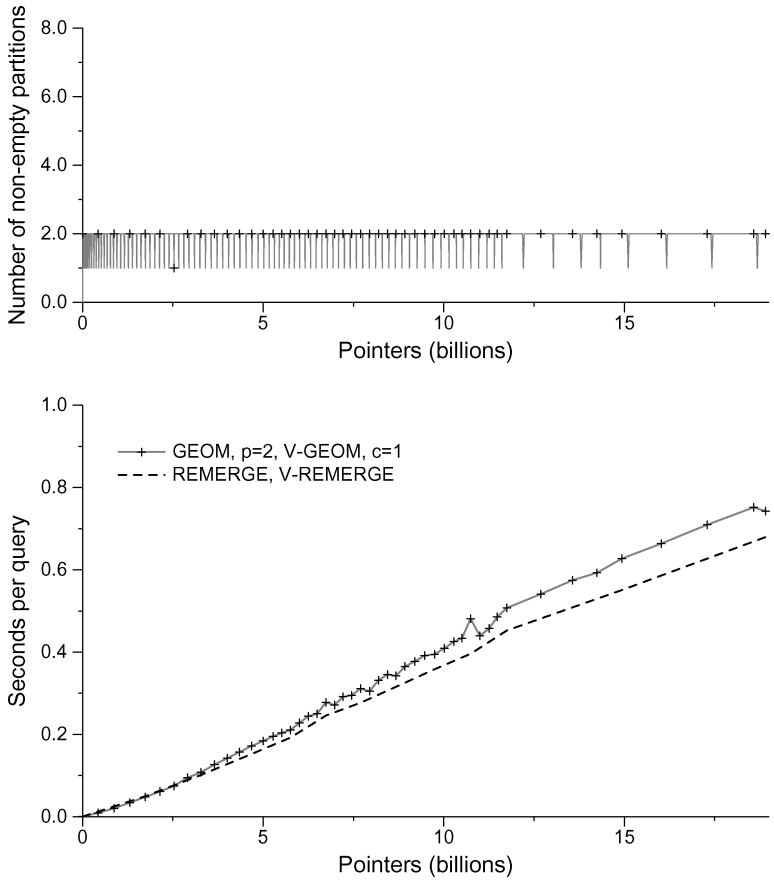

Fig. 10. Query times for a fixed-radix geometric partitioning strategy with $p = 2$, and vocabulary partitioning with one v-partition per ℓ-partition. Other details are as for Figure 9.

meaning that cache effects were limited to those we would expect to realize in practice. The dotted line shows the querying cost for the same set of data, but using an index in which each list is contiguous, as would be generated by the OFFLINE or REMERGE techniques.

The query times in Figure 9 show that the overhead arising from the use of multiple partitions is definite, but not excessive. The greatest overhead, when there are seven partitions in both the inverted lists and vocabulary, is approximately 67%. The corresponding graph for geometric partitioning with $p = 2$, $c = 1$ (Figure 10) shows an overhead of not more than 18%, and averaging around 12%.

There is a consistent relationship in Figures 9 and 10 between the query overhead (at the tick points in the lower graph) and the number of index partitions at that time (shown by the corresponding tick points in the upper graph). This relationship suggests that the additional execution time is indeed a result of the partitioning of index lists rather than any other factors, and that our models accurately predict the extent of the query overhead.

The measurements presented in Figures 9 and 10 are a worst-case estimate of the query overhead required by multiple index partitions. This overhead may be reduced by explicitly caching inverted lists in main memory, or by placing disk partitions on separate physical devices, allowing parallel disk accesses to the partitions. Improvements in query resolution speed would render geometric partitioning more attractive in comparison to the alternatives discussed in this article.

7. RELATED WORK

A range of previous work contains elements of the indexing structures we have described here. In particular, this article extends the work of Lester et al. [2005] in a number of ways: Section 5.2 presents a revised analysis, including a breakdown of the costs underlying the three key processes in index construction; Section 5.4 discusses the need for the geometric method to also be applied to vocabulary maintenance; and the experimental results in Section 6 are also more detailed, and are based on a revised implementation of geometric partitioning.

In a presentation at the same conference as our original work, Büttcher and Clarke [2005] independently described an $r = 2$ "logarithmic merge" method built on the same key insight that prompted our investigation. Büttcher and Clarke [2006] and Büttcher et al. [2006] have extended their approach to a hybrid construction scheme in which different maintenance techniques are applied to lists of different lengths in the same index, which, if coupled with the generalized geometric methods described here, can be expected to provide additional efficiency gains.

Methods based on discontiguous lists have been used in practice for some time. For example, the Lucene[2] search engine utilizes multiple partitions of nondecreasing size in order to optimize incremental indexing. The Lucene method allows, for a given radix-like parameter N, up to $N - 1$ partitions containing the same number of bufferloads, s. In contrast, geometric partitioning requires that successive merges be used to maintain a single partition containing the $s \times (N - 1)$ bufferloads. The Lucene method represents a different tradeoff between query and update efficiency, with the index potentially distributed across more partitions. Update efficiency is greater, because of the lower amount of merging required, but query resolution speed can be degraded to a greater degree than with geometric partitioning. In a similar approach, Hamilton and Nayak [2001] sketch a mechanism in which "a stack of indexes is maintained" and "new documents are put in a small index, which is periodically batch merged into a larger index."

Other work related to optimization of incremental index updates has largely focussed on inplace strategies. Cutting and Pedersen [1990] store short lists within the vocabulary during inplace index update, and Shoens et al. [1994] keep short lists within fixed-sized "buckets." Over-allocation for long lists during inplace maintenance is examined in a variety of related work [Brown et al. 1994; Shoens et al. 1994; Shieh and Chung 2005]. Clarke and Cormack [1995] discuss

[2]See `http://lucene.apache.org/`.

a remerge approach to index maintenance, with a custom space management scheme that reduces the peak disk space requirement of the scheme presented in Section 4.3.

Maintenance of B^+-tree indexes under high insertion rates has also been studied in several contexts [Graefe 2006]. Two major alternatives are represented in the literature: to buffer updates within the indexing structure [den Bercken et al. 1997], or to maintain separate indexing structures and periodically merge them, see, for example, Jagadish et al. [1997]. Buffering updates within index nodes is not efficient in incremental inverted indexing, as memory utilized for buffering updates to a B^+-tree can be more effectively allocated to buffering new postings [Cutting and Pedersen 1990]. Our proposed vocabulary maintenance scheme has similarities to that proposed by Jagadish et al. [1997], but also has several key differences. In particular, Jagadish et al. [1997] investigate the use of multiple B^+-tree indexes of nondecreasing size, analogous to the indexing scheme implemented by Lucene. As discussed above, allowing multiple partitions of the same size increases indexing efficiency relative to geometric partitioning, at the cost of greater query speed degradation. In selecting vocabulary maintenance techniques during incremental indexing, it is important to note that optimization of vocabulary maintenance is subordinate to the mechanisms used to perform inverted list maintenance, due to the significantly higher cost of the latter. Thus, the scheme used for vocabulary maintenance is informed by the choice of index maintenance strategy. One consequence of utilizing the same scheme for maintenance of both inverted lists and the vocabulary is that the B^+-tree merging can, assuming that $c = 1$ during update, operate on precisely the portion of the vocabulary that is altered during inverted list maintenance. Higher values of c trade increased querying efficiency for less exact partitioning of vocabulary entries. Our extension of geometric partitioning to vocabulary maintenance offers a combined scheme that, as shown by analysis and empirical observation, provides efficient, low-complexity update of an entire index.

8. CONCLUSIONS

We have described, analyzed, and measured a mechanism for online 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. In particular, the time required to build an index for the 426 GB `.gov` experimental collection is reduced by a factor of around seven compared to a comparable implementation of the remerge approach. The results also show that the relative gains increase with collection size, and overall construction times were within a factor of two of those achieved by a highly-optimized offline index construction scheme.

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 10% 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 also the extra cost that arises in query processing. Our work shows that online methods offer an attractive compromise among construction costs, querying costs, and access immediacy.

REFERENCES

ANH, V. AND MOFFAT, A. 2006. Improved word-aligned binary compression for text indexing. *IEEE Trans. Knowl. Data Engin. 18,* 6, 857–861.

BAEZA-YATES, R. AND RIBEIRO-NETO, B. 1999. *Modern Information Retrieval*. Addison-Wesley Longman, Reading, MA.

BAILEY, P., CRASWELL, N., AND HAWKING, D. 2003. Engineering a multi-purpose test collection for web retrieval experiments. *Inform. Process. Manag. 39,* 6, 853–871.

BAYER, R. 1972. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Inf. 1*, 290–306.

BILIRIS, A. 1992a. An efficient database storage structure for large dynamic objects. In *Proceedings of the IEEE International Conference on Data Engineering*. IEEE Computer Society, 301–308.

BILIRIS, A. 1992b. The performance of three database storage structures for managing large objects. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data (SIGMOD'92)*. ACM, New York, 276–285.

BROWN, E., CALLAN, J., AND CROFT, W. 1994. Fast incremental indexing for full-text information retrieval. In *Proceedings of the International Conference on Very Large Databases (VLDB'94)*. 192–202.

BÜTTCHER, S. AND CLARKE, C. 2005. Indexing time vs. query time trade-offs in dynamic information retrieval systems. In *Proceedings of the ACM CIKM International Conference on Information and Knowledge Management (CIKM'05)*. ACM, New York, 317–318.

BÜTTCHER, S. AND CLARKE, C. 2006. A hybrid approach to index maintenance in dynamic text retrieval systems. In *Proceedings of the European Conference on Information Retrieval (ECIR'06)*. 229–240.

BÜTTCHER, S., CLARKE, C., AND LUSHMAN, B. 2006. Hybrid index maintenance for growing text collections. In *Proceedings of the ACM-SIGIR International Conference on Research and Development in Information Retrieval (SIGIR'06)*. ACM, New York, 356–363.

CAREY, M., DEWITT, D., RICHARDSON, J., AND SHEKITA, E. 1986. Object and file management in the EXODUS extensible database system. In *Proceedings of the International Conference on Very Large Databases (VLDB'86)*. Morgan Kaufmann, 91–100.

CAREY, M., DEWITT, D., RICHARDSON, J., AND SHEKITA, E. 1989. Storage management for objects in EXODUS. In *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky, Eds. Addison-Wesley Longman, New York, 341–369.

CLARKE, C. AND CORMACK, G. 1995. Dynamic inverted indexes for a distributed full-text retrieval system. MultiText Project Tech. rep. MT-95-01, Department of Computer Science, University of Waterloo, Waterloo, Canada.

CLARKE, C., CRASWELL, N., AND SOBOROFF, I. 2004. Overview of TREC 2004 terabyte track. In *Proceedings of the 13th Text REtrieval Conference (TREC-13)*. National Institute of Standards and Technology Special Publication 500-261, Gaithersburg, MD.

COMER, D. 1979. The ubiquitous B-tree. *Comput. Surv. 11,* 2, 121–137.

CRASWELL, N. AND HAWKING, D. 2002. Overview of the TREC-2002 web track. In *Proceedings of the 11th Text REtrieval Conference (TREC-2002)*. National Institute of Standards and Technology Special Publication 500-251, Gaithersburg, MD, 86–95.

CUTTING, D. AND PEDERSEN, J. 1990. Optimizations for dynamic inverted index maintenance. In *Proceedings of the ACM-SIGIR International Conference on Research and Development in Information Retrieval (SIGIR'90)*. ACM, New York, 405–411.

DEN BERCKEN, J. V., SEEGER, B., AND WIDMAYER, P. 1997. A generic approach to bulk loading multidimensional index structures. In *Proceedings of the Conference on Very Large Data Bases (VLDB'97)*. Morgan Kaufmann, San Francisco, CA, 406–415.

GRAEFE, G. 2006. B-tree indexes for high update rates. *SIGMOD Rec. 35,* 1, 39–44.

HAMILTON, J. R. AND NAYAK, T. K. 2001. Microsoft SQL server full-text search. *IEEE Data Engin. Bull. 24,* 4, 7–10.

HARMAN, D. 1993. Overview of the first TREC conference. In *Proceedings of the ACM-SIGIR International Conference on Research and Development in Information Retrieval (SIGIR'93)*. ACM, New York, 36–47.

HEINZ, S. AND ZOBEL, J. 2003. Efficient single-pass index construction for text databases. *J. Amer. Soc. Inform. Sci. Tech. 54,* 8, 713–729.

JAGADISH, H., NARAYAN, P., SESHADRI, S., SUDARSHAN, S., AND KANNEGANTI, R. 1997. Incremental organization for data recording and warehousing. In *Proceedings of the International Conference on Very Large Databases (VLDB'97)*. Morgan Kaufmann, San Francisco, CA, 16–25.

KNUTH, D. 1973. *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd Ed. Addison-Wesley, Reading, MA.

LEHMAN, T. AND LINDSAY, B. 1989. The Starburst long field manager. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'89)*. 375–383.

LESTER, N., MOFFAT, A., AND ZOBEL, J. 2005. Fast online index construction via geometric partitioning. In *Proceedings of the ACM CIKM International Conference on Information and Knowledge Management (CIKM'05)*. ACM, New York, 776–783.

LESTER, N., ZOBEL, J., AND WILLIAMS, H. 2005. Efficient online index maintenance for contiguous inverted lists. *Inform. Process. Manag. 42,* 4, 916–933.

MOFFAT, A. AND BELL, T. A. H. 1995. In situ generation of compressed inverted files. *J. Amer. Soc. Inform. Sci. 46,* 7, 537–550.

RAMAKRISHNAN, R. AND GEHRKE, J. 2003. *Database Management Systems*, 3rd Ed. McGraw Hill, New York, NY.

SCHOLER, F., WILLIAMS, H., YIANNIS, J., AND ZOBEL, J. 2002. Compression of inverted indexes for fast query evaluation. In *Proceedings of the ACM-SIGIR International Conference on Research and Development in Information Retrieval (SIGIR'02)*. ACM, New York, 222–229.

SHIEH, W.-Y. AND CHUNG, C.-P. 2005. A statistics-based approach to incrementally update inverted files. *Inform. Process. Manag. 41,* 2, 275–288.

SHOENS, K., TOMASIC, A., AND GARCIA-MOLINA, H. 1994. Synthetic workload performance analysis of incremental updates. In *Proceedings of the ACM-SIGIR International Conference on Research and Development in Information Retrieval (SIGIR'94)*. ACM, New York, 329–338.

TOMASIC, A., GARCIA-MOLINA, H., AND SHOENS, K. 1994. Incremental updates of inverted lists for text document retrieval. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data (SIGMOD'94)*. ACM, New York, 289–300.

WILLIAMS, H. AND ZOBEL, J. 2005. Searchable words on the web. *Int. J. Digital Libraries 5,* 2, 99–105.

WITTEN, I., MOFFAT, A., AND BELL, T. C. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd Ed. Morgan Kaufmann, San Francisco, CA.

ZOBEL, J. AND MOFFAT, A. 2006. Inverted files for text search engines. ACM *Comput. Surv. 38,* 2, 1–56.

Received September 2007; revised March 2008; accepted May 2008