

A Hybrid Approach to Index Maintenance in Dynamic Text Retrieval Systems

Stefan Büttcher Charles L. A. Clarke



University of Waterloo, Canada

April 11, 2006

- 1 Introduction and Motivation
- 2 Inverted Files
- 3 Index Maintenance Strategies
- 4 Hybrid Index Maintenance
- 5 Experimental Results
- 6 Summary

Introduction and Motivation

Dynamic vs. Non-Dynamic Information Retrieval

Traditional (non-dynamic) text retrieval systems

- Take static text collection.
- Build an index for the collection.
- Run a million queries through your system.

Dynamic text retrieval systems

- Build an index for a text collection.
- Process a few queries.
- Add some new documents, remove some old ones.
- Process a few more queries.
- Repeat.

How can we *efficiently* keep the index up-to-date?

Strictly Growing Text Collections

Our contribution is limited to strictly growing text collections (no modifications to existing documents, no document deletions).

However, existing index maintenance strategies that deal with document deletions are compatible with the methods we present.

Examples of strictly growing text collections

- FBI Carnivore. (R.I.P.)
- Parliament proceedings.
- On-line libraries.
- My email folder.

Inverted Files

Index Structure: Inverted Files

An inverted file realizes a mapping from each term in a text collection to all its occurrences (*inverted list* or *posting list*).

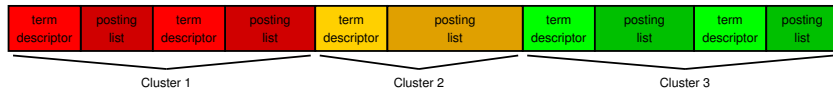
In all our experiments, inverted files contain full positional information (no document-level or frequency indices); terms are stored in the index in unstemmed form, sorted lexicographically.

Unlike most other systems, we do not maintain an explicit vocabulary containing the exact on-disk locations of all posting lists (This is an important performance aspect: The GOV2 text collection, depending on the exact tokenization method, contains between 40 and 80 million distinct terms).

But without a vocabulary, how can the posting list for a given term be found in the index?

Inverted Files in Detail

The inverted file is divided into clusters of posting lists:

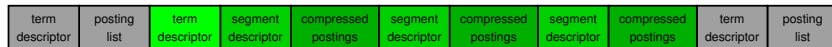


For each cluster, only the descriptor for the first term makes it into the vocabulary and is loaded into memory. Each cluster contains about 64 KB of data, each descriptor consumes 28 bytes. Size of the vocabulary: $\approx 0.05\%$ of the total index size.

⇒ Very small memory footprint; comes at the cost of a tiny performance loss at query time (disk seek plus reading 64 KB instead of disk seek only).

Inverted Lists in Detail

Each inverted list consists of a number of list segments:



Each segment contains at most 2^{16} postings and is stored in compressed form, to decrease disk I/O.

During query processing, all posting list segments for each query term are loaded into memory and decompressed as needed.

Index Maintenance Strategies

Buffering Updates in Memory

Virtually every index maintenance strategy for growing text collections follows the same general pattern:

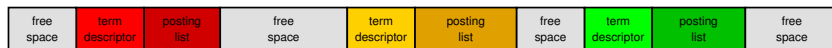
- Buffer data from incoming documents in memory, building an in-memory inverted file.
- Once the size of the in-memory index reaches a certain threshold, all data are transferred from memory to disk and somehow combined with existing on-disk index data.

Index maintenance strategies only differ in their implementation of “somehow”.

The in-memory index can easily be organized in such a way that it can be queried, resulting in a fully dynamic search system.

In-Place Update

Idea: Leave gaps in the on-disk index for future postings; relocate posting lists if not enough free space at current position.

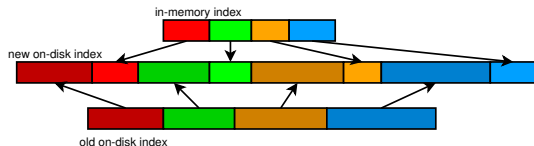


Two problems with in-place update:

- Relocations destroy the lexicographical ordering of on-disk posting lists. \Rightarrow An explicit vocabulary is needed.
- Relocations imply disk seeks. For an index containing 50 million terms, relocating only 0.1% of the posting lists requires around 100,000 disk seeks (about 16 minutes). \Rightarrow Slow!

Merge-Based Update: Immediate Merge

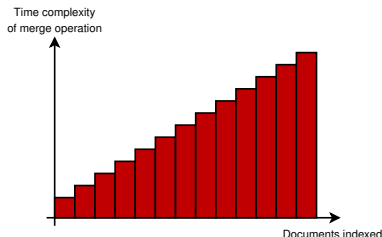
Merge-based index maintenance strategies do not leave free space in the inverted file. Whenever main memory is exhausted, a new on-disk inverted file is created by merging the existing one with the in-memory data (different colors represent different terms):



Immediate Merge is more efficient than most in-place update strategies because of its largely sequential disk access pattern. Almost no disk seeks are necessary.

Merge-Based Update: Immediate Merge

The time complexity of the n -th merge operation is linear in the current size of the on-disk index: $\Theta(n \cdot M)$.



The total index maintenance overhead is $\Theta\left(\frac{C^2}{M}\right)$, where C is the size of the collection and M the size of the in-memory buffers.

Merge-Based Update: Logarithmic Merge

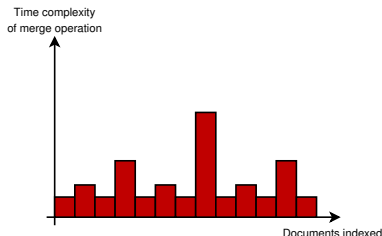
With Logarithmic Merge, the search engine may maintain more than one on-disk index at the same time. Every on-disk inverted file gets a label, its *generation number*:

- An inverted file that is created directly from in-memory data is of generation 0.
- An inverted file that is created by merging two inverted files of generation n is of generation $n + 1$.

Whenever there is a collision (two inverted files of the same generation), inverted files of the same generation are merged until there are no more such collisions.

Merge-Based Update: Logarithmic Merge

The average time complexity of the n -th merge operation is sub-linear in the current size of the on-disk index.



The total index maintenance overhead is $\Theta\left(C \log \frac{C}{M}\right)$, where C is the size of the collection and M the size of the in-memory buffers. However: Additional disk seeks at query time.

Deferring the Update to the File System: Hans Reiser

Hans Reiser, designer of the Reiser4 file system:

Storage Layers Above the FS:

A Sure Symptom The FS Developer Has Failed

The very existence of a storage layer above the filesystem means that the filesystem team at an OS vendor failed to listen to someone, and that someone was forced to go and implement something on their own.

(<http://www.namesys.com/whitepaper.html>)

We tried to find out whether his bold performance claims are realistic or not.

Performance of File-System-Based Index Maintenance

We built an index for TREC disks 4+5 (528,000 documents, around 2 GB of text), comparing the performance of Reiser4 as a storage layer (1 posting list per file) with that of Immediate Merge.

Size of in-memory buffers: 256 MB

Method	Indexing time	Index size	Num. of files
Reiser4	22 minutes	1.7 GB	1.2 million
Immediate Merge	13 minutes	1.2 GB	9

Size of in-memory buffers: 128 MB

Method	Indexing time	Index size	Num. of files
Reiser4	40 minutes	1.7 GB	1.2 million
Immediate Merge	17 minutes	1.2 GB	9

Hybrid Index Maintenance

In-Place versus Merge-Based Update

Advantages and disadvantages of in-place and and merge-based update strategies:

- **In-place update** is very good at dealing with long posting lists, but lousy at dealing with lots of short lists: Many relocations, many disk seeks. As long as we have only few lists, we are fine, though.
- **Merge-based update** is very good at dealing with lots of short posting lists (no extra disk seeks, because of sequential disk access), but lousy at dealing with very long lists, as they have to be copied over and over again. Logarithmic Merge reduces the copy overhead, but the general problem remains the same.

Some Statistics from GOV2

The GOV2 text collection is the result of a webcrawl of the .gov domain and it used in the TREC Terabyte track.

- Total size of collection: 426 GB.
- Number of documents: $25.2 \cdot 10^6$.
- Number of tokens: $41.7 \cdot 10^9$.
- Number of distinct terms (after stemming):
 $47.3 \cdot 10^6$ (e.g., “tharapiwattananon”, “gagtggctagaga”).
- Number of long posting lists ($> 10^6$ entries):
3,132 (0.0066%).
- Number of postings in long lists ($> 10^6$ entries):
 $37.6 \cdot 10^9$ (79.5%).

Combining In-Place and Merge-Based Update

Idea: Take the best from both worlds. Use in-place update for long lists and merge-based update for short lists.

Because implementing a decent in-place update strategy is rather difficult and we wanted to get our results quickly, we chose to implement the in-place part by deferring it to the file system (Thanks, Hans!).

As a file system, we used ext3, primarily because Reiser4 is not officially part of the Linux kernel yet (Sorry, Hans!).

Definition of Hybrid Index Maintenance

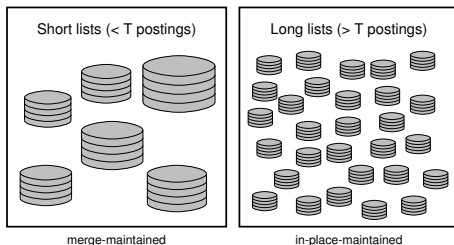
Before the system is started, a parameter T (the threshold value) needs to be chosen.

Hybrid index maintenance behaves like merge-based index maintenance. Only difference: Whenever, during a merge operation, a posting list is encountered that contains more than T postings, it is defined to be *long* and moved to the in-place part of the on-disk index. From that point on, it is always updated in-place.

Every *long* list has its own file in the file system.

Hybrid index maintenance can be combined with every merge-based strategy (Immediate Merge, Logarithmic Merge, ...).

On-Disk Index Layout



- Merge part: Few, larger inverted files (each containing many different terms).
- In-place part: Many, smaller inverted files (each containing one single term).

Partial Flush

The main motivation behind hybrid index maintenance is to avoid copying these very long posting lists during merge operations, without introducing too many additional disk seeks.

The best way to achieve this is to not perform merge operations. Or at least delay them.

Partial flush: Whenever main memory is exhausted, only transfer postings for terms with *long* posting lists to disk (i.e., only perform in-place updates, but no merging). Only start a merge operation if postings for long lists contribute less than 15% to the memory consumption.

This reduces the number of merge operations by a factor 2-3.

Experimental Results

Experimental Setup

All experiments were conducted on a single PC based on an AMD Athlon64 3500+ CPU (2.2 GHz) with 2 GB of RAM.

The index was stored on a 7,200-rpm SATA hard drive. The text collection was read from a RAID-0 built on top of 2 disks.

The command sequence used in all experiments was a mixed update/search sequence, simulating a dynamic search environment (about 1 search query per 18,000 document insertions).

The underlying text collection was GOV2. The queries were randomly selected from the TREC Terabyte ad-hoc topics (2004 and 2005).

Non-Hybrid Strategies

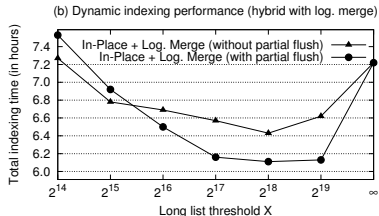
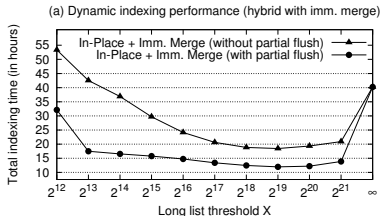
Indexing GOV2 with 1024 MB for the in-memory index; processing search queries in parallel.

	No Merge	Imm. Merge	Log. Merge
Total indexing time	4h01m	40h14m	7h14m
Indexing throughput	106.1 GB/h	10.6 GB/h	59.0 GB/h
Average time per query	5.028 sec	2.840 sec	3.011 sec

No Merge is a dynamicized version of off-line index construction (on-disk inverted files are only merged at the very end).

Hybrid Index Maintenance Performance

Total time to build an index for GOV2:

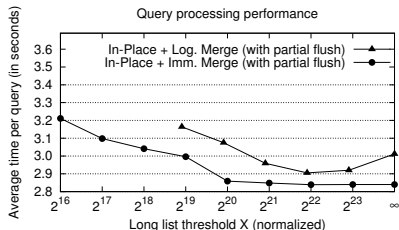


Hybrid index maintenance increases the overall index maintenance performance of both Immediate Merge and Logarithmic Merge. With partial flush enabled, the savings are even greater.

Hybrid Query Processing Performance

Average search engine response time per query:

(run BM25, report top 20 documents; performance as of Oct 2005)



For Immediate Merge, $T = \infty$ (the non-hybrid base strategy) achieves the highest query throughput (no disk seeks). For Logarithmic merge, the hybridization results in better query performance (reducing disk seeks).

Summary

- We have proposed a hybrid index maintenance strategy, based on a distinction between short and long posting lists.
- Short lists are maintained according to a merge strategy.
- Long lists are maintained in-place, by deferring the update to the file system.
- Hybrid index maintenance improves indexing performance (by more than 300% in the case of Immediate Merge; by about 20% in the case of Logarithmic Merge).
- Query processing performance stays about the same or even improves a bit (in the case of Logarithmic Merge).

Advertising

- This is all part of the Wumpus search engine (in a similar form – see papers and/or code for details).
- Free code (GPL) and free papers (not GPL) are available at:
<http://www.wumpus-search.org/>

The End

Thank You!