Scalable Online Index Construction with Multi-core CPUs

Hiroyuki Yamada  Motomichi Toyama

Department of Information and Computer Science
Keio University, Yokohama, Japan
Email: hiroyuki@db.ics.keio.ac.jp,
toyama@ics.keio.ac.jp

Abstract

Inverted index is a core element of current text retrieval systems. They can be dynamically constructed using online indexing approaches in the environment in which even small delays in timeliness cannot be tolerated, and the index must always be queryable and up to date. Recently, efficient online index construction schemes have been proposed; however, previous works have not focused on scalability with the modern commodity hardware resources such as multi-core CPUs. In this paper, we propose a scalable online index construction method that better utilizes multi-core CPUs. Using experiments on 30 GB of web data, we demonstrate the efficiency of our method in practice, showing that it dramatically reduces online index construction time without sacrificing query performance.

Keywords: Information Retrieval, Text Databases, Inverted Index, Online Index Construction, Index Maintenance

1 Introduction

Inverted index is a core element of current text retrieval systems. It is a data structure that maps a word, or atomic search item, to the set of documents, or set of indexed units, that contain that word – its postings. An individual posting may be a binary indication of the presence of that word in a document, or may contain additional information, such as its frequency in that document and an offset for each occurrence, required for various non-boolean search algorithms. Inverted index can be constructed using either off-line approaches or online approaches. In off-line approaches, 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. On the other hand, online approaches are used when even a small delay in timeliness cannot be tolerated and the index must always be queryable and up to date. In the current search environment, online approaches are more general and there have been various methods proposed recently.

Basic techniques to support document insertions into an existing index usually follow a standard scheme: two indexes are maintained, one in memory, the other on disk. Postings for new documents are accumulated in main memory until it is exhausted, at which point they are transferred to disk and combined with existing on-disk data. This operation can be performed by an in-place update scheme (17) or by merging the old index with the new data, resulting in a new index that supersedes the old one (9). More efficient merging techniques are proposed (3; 10), which allows a controlled number of on-disk indexes to increase indexing performance without decreasing query processing performance excessively.

In recent years, commodity hardware have much more resources than few years ago. Regarding CPUs, major vendors currently have dual-core and quad-core CPUs, and some vendor has an eight-core CPU. Also, a lot of memory is available in 64 bit operating systems. On the current desktop/server machine, dual-core CPU and 4 GB memory is a very common setting and multiple dual/quad-cores with more main memory is often used in servers. However in this situation, previous online index construction methods are not designed to scale for such modern hardware resources, that is, index construction time does not reduce as the number of available cores increases even though scalability is one of the most important factors in current database systems.

In this paper, we propose a scalable online index construction method which better utilizes multi-core CPUs. The principle of our new method is that multiple in-memory indexes are parallelly constructed by multiple cores, and aims to reduce the total construction time. To the best of our knowledge, this is the first study focused on exploiting multi-core CPUs in online index construction. Our experiments show that online index construction using this approach significantly reduces index construction time without sacrificing query performance.

The rest of this paper is organized as follows: we begin by describing prior work on online index construction in Section 2. In Section 3, we provide new inverted index data structure which in-memory indexes can be parallelly constructed with, and introduce some optimization techniques to get better performance. Section 4 provides experimental results and their evaluation and we give a discussion about the proposed method and related work in Section 5. Finally, Section 6 gives the conclusion and future work.

2 Background

2.1 Inverted Index Structures

Inverted 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 postings (also sometimes known as pointers), together with a range of ancillary information, which can include within-document frequencies and a subsidiary list of positions within each document at which that term appears (12). A range of compression techniques have been developed for inverted lists (1; 13; 18; 12; 20), 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 postings in each inverted list in document ID order, and is referred to as being document sorted. It is the index organization that we consider in this paper. Other index orderings are frequency sorted and impact sorted. None of the online mechanisms we describe in this paper apply to these other forms of index organization.

A retrieval system processes queries by examining the postings for the query terms, and using them to calculate a similarity score that estimates the likelihood that the document matches the query. This process requires that all terms in the collection be indexed, with the exception of a small number of common terms that carry little information (stop-words). 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.

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

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

2.2 Index Merging

As described in the previous section, various merging techniques between in-memory indexes and on-disk indexes have been proposed. Figure 1 shows the major merging methods, which are described in the following subsections.

2.2.1 Immediate Merge

The first merge strategy has been proposed by Lester et al. (9). The indexing system maintains one on-disk and one in-memory index. As soon as main memory is full, the in-memory postings are merged with the existing on-disk index, creating a new index. The old index is deleted. This strategy minimizes the number of disk seeks necessary to fetch a posting list. Its disadvantage is that for every merge operation the entire index has to be scanned. Thus, the number of disk operations necessary to index the entire collection is quadratic in the size of the text collection.

2.2.2 No Merge

The second strategy does not perform any merge operations. When memory is full, postings are sorted and written to disk, creating a new on-disk sub-index. On-disk indexes are never merged. When the posting list for a given term has to be retrieved from the index, sub-lists are fetched from all sub-indexes. The advantage of No Merge is its high indexing performance (linear number of disk operations). Its disadvantage is that fetching a posting list requires $O(n)$ disk seeks, where $n$ is the size of the text collection.

2.2.3 Geometric Partitioning

The two strategies described so far represent the two extremes. The third strategy is a compromise: a
newly created on-disk sub-index is sometimes merged with an existing one, but not always.

Lester et al. (10) propose a scheme that breaks the index into tightly controlled number of partitions. They introduce a key parameter \( r \) (usually, \( r = 2 \) or \( r = 3 \) is chosen). If main memory can hold \( b \) postings, then the \( k \)th partition contains not more than \((r-1)r^{k-1}b\) postings. In addition, at level \( k \) the partition is either empty, or contain at least \( r^{k-1}b \) postings. Whenever the creation of a new on-disk index leads to a situation where there are more than \((r-1)r^{k-1}b\) postings in \( k \)th partition, they are merged into a new index and dispatched to the appropriate partition. This strategy is referred to as Geometric Partitioning. Lester et al. (10) gives a cost analysis of this strategy. Büttcher and Clarke (3) also propose a similar strategy referred to as Logarithmic Merge and it is regarded as ‘\( r = 2 \) Geometric Partitioning’ (11).

Geometric Partitioning and Logarithmic Merge strategies are designed for growing text collections and offer better index maintenance performance than Immediate Merge. The disadvantage is that query processing performance is worse, as posting list of each term may spread across a number of on-disk sub-indexes.

2.2.4 Other Strategies

In-place update strategy involves minimizing the change to the index at each stage by, whenever possible, writing new postings at the end of the existing lists, and related works have been broadly researched in (5; 2; 8; 14; 15). Büttcher et al. (4) proposed hybrid strategy based on a distinction between short and long posting lists, which maintains short posting lists following a merge-based approach, while long lists are updated in-place. Also, Guo et al. (6) proposed another merge strategy that can dynamically adjust the sequence of sub-index merge operations during index construction, and offers better query processing performance than previous methods with support for instantaneous document deletions.

3 Parallel Construction of Multiple In-memory Indexes

We have discussed the index update strategies for continuously growing text collections in the previous sections. Indexing with Geometric Partitioning is very efficient in terms of disk I/O and practically one of the best option in a current dynamic environment. However, in-memory part is still time-consuming with in-memory index construction, which includes parsing, index compression and in-memory inversion. Also, it has no focus on scalability with modern hardware resources, that is, its performance does not increase as hardware resource increases.

To deal with the issue, we focus on Geometric Partitioning with multi-core CPUs and propose a new method to achieve scalable online index construction. We first propose a novel data structure to achieve parallel construction of multiple in-memory indexes, and then, introduce some optimization techniques enhancing its performance. In summary, our main contributions are as follows:

1. We propose a novel data structure of inverted index which enables parallel construction of multiple in-memory indexes. Also, we present an intersection algorithm for the proposed data structure briefly.

2. We propose some optimization techniques called Out-of-Order Merging and Lazy Merging, which enhance the parallel construction performance.

3. We present the experimental results on a crawled 30 GB web-based document collection and show that the proposed method dramatically reduces online index construction time without sacrificing query performance.

3.1 Multiple In-memory Indexes

In order to achieve scalable index construction with multi-core CPUs, constructing multiple in-memory indexes parallelly by cores seems to be a good approach. This approach divides an time-consuming in-memory index construction process into the number of cores available. Figure 2 shows parallel construction of multiple in-memory indexes and on-disk merging in a timeline view. As shown in Figure 2, each in-memory process constructs each index and merging occurs when in-memory process completes its construction. However, if using a naive approach, parallel construction of multiple in-memory indexes has difficulty in managing document sorted inverted lists. In this paper, we propose a solution for this problem.

![Figure 2: Parallel construction of multiple in-memory indexes and on-disk merging in a timeline view.](image)

3.2 Inverted Index with Two-level Document ID

To deal with the problem described above, we propose a novel inverted index data structure with two-level document ID, which manages document IDs in inverted lists in two level with Global Sequence Number (GSN) and local document ID. With this scheme, in-memory processes can independently proceed with local document ID as usual way as long as an unique GSN is assigned to those in-memory indexes. Also, by adding the GSN to the beginning of the local document IDs, finally created inverted lists are ordered globally by GSN and locally by local document ID as shown in Figure 3. One thing to keep in mind when dealing with parallel in-memory processes is that the merge operation must be coordinated between the processes not to proceed concurrently, so that inverted lists are properly created in ascending GSN order.

To meet the memory requirement, the specified memory is split into the number of in-memory processes. For example, to make 8 in-memory processes parallelly runnable with 1 GB memory as a buffer, 128 MB memory is assigned to each in-memory process.

3.3 Out-of-Order Merging

By assigning a GSN to an in-memory index just before merging, merging can be done in an out-of-order manner and this gives better performance. That is, there is no order between in-memory indexes. As soon as any in-memory index exceeds a specified threshold, merging for the index can be executed immediately without any stall when no other processes are
in merging. Here we describe the basic steps for each process in constructing inverted index with Out-of-Order Merging.

1. For each in-memory process,
   (a) Postings for new documents are accumulated in the in-memory index.
   (b) When the in-memory index size reaches to the threshold,
       i. Assign a GSN to the in-memory index.
       ii. Merge the in-memory index with the existing on-disk indexes. (wait while another index is in merging.)
   (c) After the merging finishes, go back to step 1a.

3.4 Lazy Merging

As we described, each process has its own split buffer for in-memory index. Its size is relatively small compared with the buffer size in a single in-memory process, so frequent merging is avoidable and incurring a lot of disk I/O as a result. We propose an optimization technique called Lazy Merging for this issue. It prepares a queue first and puts the in-memory index to the queue as soon as it exceeds the buffer size. Then the process can proceed another construction of in-memory index with a new buffer. When the queue becomes full, then flushing proceeds with multi-way merging the queued in-memory buffers with existing on-disk indexes. This method avoids the frequent small sized merging and saves a lot of disk I/O. Figure 4 shows conceptual view of Lazy Merging. It is actually implemented more efficiently without memory copy and consumption of unused buffers. It is also a little similar to the well-known 'double buffering' technique, but our method defers I/O operation and merging occurs in batch afterwards, and buffers and in-memory processes are independent so in-memory process proceeds index construction as long as there is an available buffer in the pool.

To meet the memory requirement, we split the memory into the number of buffers being used, so that this works efficiently without extra memory consumption. From our experiments, when we have $n$ in-memory processes, index buffer should be split into $2n$ and size of the queue should be set in $n$ to achieve better performance. For example, When we have 1 GB buffer available for 8 in-memory processes, the buffer had better to be split into 16 and buffer queue is set in size 8. The following describes details about the index construction steps with Lazy Merging.

1. Prepare an empty queue for in-memory index
2. For each in-memory process,
   (a) Postings for new documents are accumulated in in-memory index.
   (b) When the in-memory index size reaches to the threshold, assign a GSN to the in-memory index and push it into the queue.
       i. If the queue is not full, go back to step 2a.
       ii. If the queue is full, go to next step.
   (c) Multi-way merge the queued indexes with the existing on-disk indexes.
   (d) After merging finishes, the queued buffers are cleared and accept new buffers to be pushed, and go back to step 2a.

3.5 Intersection Algorithm

Intersection algorithm in querying, such as in boolean queries and phrase queries, must be modified to adapt the inverted index with two-level document ID created by the proposed method. Algorithm 1 shows pseudo intersection algorithm between two posting lists. The main difference is that GSN checking operation is added before document ID checking operation.

4 Experiments

We have experimented with a crawled 30 GB web-based document collection, which consists of 22.7 million documents, and measured the index construction time and the query time. All experiments are performed on linux based on a dual-processor Quad-Core Xeon 5345 machine with 7,200-rpm SATA hard drive, and it has 8 GB of RAM with no significant other processes running at the time of the experiments. We adjusted the number of enabled cores by setting `/sys/devices/system/cpu/cpu[CPU ID]/online` parameter through the linux file system. In this experiments, we focused on Geometric Partitioning ($r=2$) as an online index construction method. Inverted lists include term frequency and positional indexes for each document ID and document IDs and
Algorithm 1 INTERSECT(p1, p2)
1: \(\text{answer} \leftarrow \epsilon\)
2: \(\text{while } p1 \neq \text{NIL} \text{ and } p2 \neq \text{NIL} \text{ do}\)
3: \(\text{if } \text{GSN}(p1) = \text{GSN}(p2) \text{ then}\)
4: \(\text{ADD} \text{(answer, GSN}(p1))\)
5: \(\text{while hasDoc}(p1) \text{ and } hasDoc(p2) \text{ do}\)
6: \(\text{if docID}(p1) = docID(p2) \text{ then}\)
7: \(\text{ADD} \text{(answer, docID}(p1))\)
8: \(p1 \leftarrow \text{nextDoc}(p1)\)
9: \(p2 \leftarrow \text{nextDoc}(p2)\)
10: \(\text{else}\)
11: \(\text{if docID}(p1) < docID(p2) \text{ then}\)
12: \(p1 \leftarrow \text{nextDoc}(p1)\)
13: \(\text{else}\)
14: \(p2 \leftarrow \text{nextDoc}(p2)\)
15: \(\text{end if}\)
16: \(\text{end if}\)
17: \(\text{end while}\)
18: \(p1 \leftarrow \text{nextGSN}(p1)\)
19: \(p2 \leftarrow \text{nextGSN}(p2)\)
20: \(\text{else}\)
21: \(\text{if } \text{GSN}(p1) < \text{GSN}(p2) \text{ then}\)
22: \(p1 \leftarrow \text{nextGSN}(p1)\)
23: \(\text{else}\)
24: \(p2 \leftarrow \text{nextGSN}(p2)\)
25: \(\text{end if}\)
26: \(\text{end if}\)
27: \(\text{end while}\)

positional indexes are stored as the differences from the previous number. Also, GSN gaps are taken in inverted index with two-level document ID. They are also compressed by variable-byte codes (12; 20), which is a well-known and very effective compression method adopted in many search engines. The elapsed times are presented for index construction including all parsing, index compression, indexing, and list merging phases. The query time includes searching vocabularies, fetching lists from the disk and lists intersection, but doesn’t include other processes such as scoring results, fetching documents and snippet generation, because those processes are not affected by the proposed data structure.

We have implemented the system from scratch in C++. We used std::map in STL for the in-memory index and Berkeley DB-like original B+-tree index compression methods such as internal structures managed by the library. So indexing with 1 GB buffer actually uses more memory than specified. Also, we applied Geometric Partitioning to both vocabularies and postings, which is described in (11) as a more scalable indexing method.

Figure 6 shows the index construction time by our proposed method, using 1 GB buffer with Out-of-Order Merging for the 30 GB document collection, for the range of the number of enabled cores. There are three lines, one at the top shows the construction time with the existing method and the other two at the bottom show the construction time with the proposed method with or without Lazy Merging. These results indicate that the proposed method reduces the construction time dramatically as the number of enabled cores increases, even though the existing method has no effect on the construction time with 8 cores enabled. The reduction rate for the proposed method degrades gradually, but Lazy Merging technique relax this degradation and additional performance gain is obtained as expected. This degradation is considered to be due to stall of threads, which finish in-memory part and wait for merging event to finish. Thus Lazy Merging, which reduces disk I/O incurred with merging, becomes a more effective technique as the number of enabled cores increases.

Figure 7 shows the query time with two-term phrase queries by a single thread for the inverted index with two-level document ID, which is created by 8 threads, and conventional inverted index. It is the sum of the query times for all existing on-disk indexes. It depends on the exiting number of indexes at the time experiments performed, so we measured the query time for the range of the size of documents indexed. We did not measure the query time for in-memory indexes in these experiments because it is relatively small compared with the query time for on-disk indexes and thus ignorable. And also, we measured three types of queries, highly frequent, moderately frequent and less frequent, for the index of 30 GB data. This is because that the posting list of a less frequent query is likely to be more occupied by GSNs, and this might affect the query performance badly. Figure 7 indicates that inverted index with two-level document ID does not slow down the query time, and the size of the inverted lists created by this scheme is almost the same as the size of the conventional inverted lists as shown in Table 1. We believe the reason for this is the followings. The document IDs between GSNs are relatively smaller than the document IDs assigned with the usual way and the gaps between document IDs become smaller. Also, the gaps between GSNs usually do not become too large. So, the compression methods such as variable-byte codes work effectively for those small numbers, and thus the size of the proposed inverted lists does not become much larger.

5 Discussion

In this paper, we focused on an online index construction method with multi-core CPUs and single disk. Applying this method to a machine with multiple disks and getting more scalable index construction is going to be a next challenge.

Utilizing multiple cores/CPU's and multiple disks can also be considered in parallel information retrieval (7; 16; 19). It can also be distributed information retrieval with virtual hosts by virtualization technology. So, a machine with multiple cores/CPU's and multiple disks is regarded as multiple processors or

1http://luxio.sourceforge.net/ - Yet Another Fast Database Manager.

Figure 5: System overview with multiple in-memory indexes.

The specified buffer for indexing counts the amount consumed for actual data of vocabularies and postings. That is, it doesn’t include library dependent data such as internal structures managed by the library. So indexing with 1 GB buffer actually uses more memory than specified. Also, we applied Geometric Partitioning to both vocabularies and postings, which is described in (11) as a more scalable indexing method.
Figure 6: Index construction time for the 30 GB collection on dual processor Quad-Core Xeon 5345 with 1 GB memory buffer, for the range of the number of enabled cores (1,2,4,8).

Figure 7: Query time with two-term phrase queries by a single thread for inverted index with two-level document ID (with GSN), which is created by 8 threads, and conventional inverted index, for the range of the size of documents indexed.

<table>
<thead>
<tr>
<th></th>
<th>size of inverted lists with two-level document ID</th>
<th>size of conventional inverted lists</th>
<th>increase rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>highly frequent</td>
<td>184307990 (bytes)</td>
<td>184306856 (bytes)</td>
<td>0.00062 (%)</td>
</tr>
<tr>
<td>moderately frequent</td>
<td>34019764 (bytes)</td>
<td>34018979 (bytes)</td>
<td>0.0023 (%)</td>
</tr>
<tr>
<td>less frequent</td>
<td>13444111 (bytes)</td>
<td>13441946 (bytes)</td>
<td>0.016 (%)</td>
</tr>
</tbody>
</table>

Table 1: Size of inverted lists of selected phrase queries which created by two-level document ID scheme and by a conventional scheme for 30 GB document collection.
multiple hosts, and document partitioning technique can be applied to get better scalability. The method described in this paper still can be applied in each processor which likely has multiple cores (2–4 cores) to handle a lot of operations such as querying and accepting TCP connections when indexes are owned by server processes for instance.

Document partitioned inverted indexes can be constructed independently using the split resources of modern hardware as described above, however, independent online index construction increases the number of indexes per CPU. For example, let’s consider that we have a machine with Quad-Core CPU and 4 disks and 100 GB data to be indexed with 1 GB memory available for a buffer. We can have 4 logically independent processors with 1 core and 1 disk each. In this setting, indexing with Geometric Partitioning is parallelly proceeded in each processor for 25 GB data with 256 MB memory buffer, and the index construction time can be reduced by factor of 4 compared with the time by a single core. However, the number of indexes created is also 4 times larger (4 * log(25 G/256 MB) = 28), and this degrades query performance excessively in exchange for reduction of index construction time.

We believe that applying the proposed method to multiple disks and creating the same number of indexes as a single processor does by collaborating on-disk merging could be another interesting research area in online index construction with modern hardware. Of course, that kind of machine can be clustered, and document partitioned technique is applied to get more scalability that cannot be obtained by a single machine.

Another issue about multi-core CPUs is the cache consciousness. Even though we focused on multi-core CPUs in this paper, we didn’t really take into consideration cache consciousness in the in-memory process. It could be another very important issue in terms of processing with multi-core CPUs, and utilizing cores with cache consciousness to shorten the time in in-memory process, then the total index construction time could be reduced.

6 Conclusion and Future Work

In this paper, we have proposed a novel method for online index construction using multi-core CPUs, which is based on the principle of parallel construction of multiple in-memory indexes for scalable online index construction. Our experimental evaluation has shown that the proposed method with some optimizations reduces index construction time close to by a factor of 4 with 8 cores without loss of query time.

We believe that optimization of the method is a promising area for future work. Also, applying multiple disks to the method to get more scalable index construction is another work in the future.

References


