Index Construction - IR weekly report 4

hello underworld

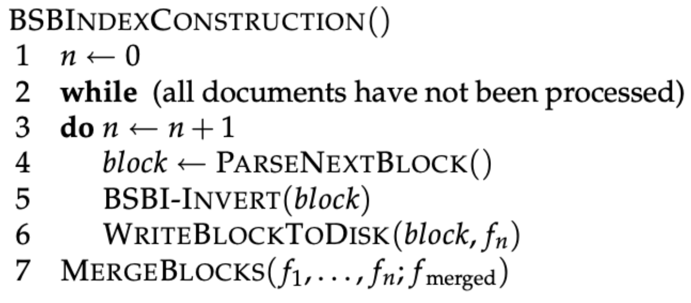
22nd May 2021

1 Introduction

Here, I just spend more than a week understanding several types of methods for index construction. No new concept is introduced, so, I directly make the notes for these methods except for some hardware basics as following:
(1):Access to data in memory is much faster than access to data on disk.
(2):When doing a disk read or write, it takes a while for the disk head to move to the part of the disk where the data are located.To maximize data transfer rates, chunks of data that will be read together should therefore be stored contiguously on disk.
(3):Operating systems generally read and write entire blocks. Thus, reading a single byte from disk can take as much time as reading the entire block.
(4):We call the part of main memory where a block being read or written is stored a buffer
(5):Data transfers from disk to memory are handled by the system bus, not by the processor. This means that the processor is available to process data during disk I/O. We can exploit this fact to speed up data transfers by storing compressed data on disk.

2 Blocked Sort Based Indexing(BSBI)

The process of BSBI is largely 4 steps as following:
(1)We need the dictionary (which grows dynamically) in order to implement a term to termID mapping
(2) segments the collection into parts of equal size(the size of a block)
(3) sorts the termID–docID pairs of each part in memory
(4) stores intermediate sorted results on disk
(5) merges all intermediate results into the final index.
Here is the algorithm for this process in figure 1. And there are 3 points should



```
BSBIndexConstruction()
1  $n \leftarrow 0$
2  while (all documents have not been processed)
3  do $n \leftarrow n + 1$
4      $block \leftarrow$ ParseNextBlock()
5      BSBI-Invert($block$)
6      WriteBlockToDisk($block, f_n$)
7  MergeBlocks($f_1, \ldots, f_n; f_{\mathrm{merged}}$)
```

Figure 1: algorithm for BSBI

be cared:
(1):The algorithm parses documents into termID–docID pairs and accumulates the pairs in memory until a block of a fixed size is full ($PARSENEXTBLOCK()$ in figure 1). We choose the block size to fit comfortably into memory to permit a fast in-memory sort.
(2)How to merge all the postings list in each block faster?The algorithm simultaneously merges all blocks into one large merged index.In each iteration, we select the lowest termID that has not been processed yet using a priority queue or a similar data structure. All postings lists for this termID are read and merged, and the merged list is written back to disk.
(3):How expensive is BSBI? Its time complexity is $(TlogT)$ because the step with the highest time complexity is sorting and T is an upper bound for the number of items we must sort

3 Single-pass in Memory Indexing

Three differences between the SPIMI and BSBI are:
(1)SPIMI generates separate dictionaries for each block – no need to maintain term-termID mapping across blocks.
(2)SPIMI doesn't sort and accumulates postings in postings lists as they occur.
(3)SPIMI could make use of compression of terms and postings to be more efficient.
However, the process of merging all index in each block is analogous to BSBI. Here is the algorithm for the process of SPIMI in figure 2 :

```
SPIMI-INVERT($token\_stream$)
  $output\_file$ = NEWFILE()
  $dictionary$ = NEWHASH()
  while (free memory available)
  do $token \leftarrow next(token\_stream)$
      if $term(token) \notin dictionary$
        then $postings\_list$ = ADDTODICTIONARY($dictionary, term(token)$)
        else $postings\_list$ = GETPOSTINGSLIST($dictionary, term(token)$)
      if $full(postings\_list)$
        then $postings\_list$ = DOUBLEPOSTINGSLIST($dictionary, term(token)$)
      ADDTOPOSTINGSLIST($postings\_list, docID(token)$)
  $sorted\_terms \leftarrow$ SORTTERMS($dictionary$)
  WRITEBLOCKTODISK($sorted\_terms, dictionary, output\_file$)
  return $output\_file$
```

Figure 2: algorithm for SPIMI

4 Distributed Indexing

For web-scale indexing (don't try this on your PC):must use a distributed computing cluster. Obviously, this scale of the index is several orders of magnitude larger.Here are three key points for the method.
(1)Maintain a master machine directing the indexing job considered "safe".
(2)Break up indexing into sets of (parallel) tasks.
(3)Master machine assigns each task to an idle machine from a pool.

Actually, the distributed indexing is also named as Mapreduce which represents two large steps in this method:
(1)Map phase:mapping splits of the input data to key-value pairs.This is the same parsing task we also encountered in BSBI and SPIMI, and we therefore call the machines that execute the map phase parsers.
(2):Reduce phase:an inverter collects all (term,doc) pairs (= postings) for one term-partition.Sorts and writes to postings lists.
Here is the schema for MapReduce:

$$map : input- > list(k, v)$$
$$reduce : (k, list(v))- > output$$

And the instantiation of the schema for index construction:
map: collection → list(termID, docID)
reduce: ((termID1, list(docID)), (termID2, list(docID)), . . .) → (postings list1, postings list2, . . .)

This is a simple process of distributed index on the book in figure:

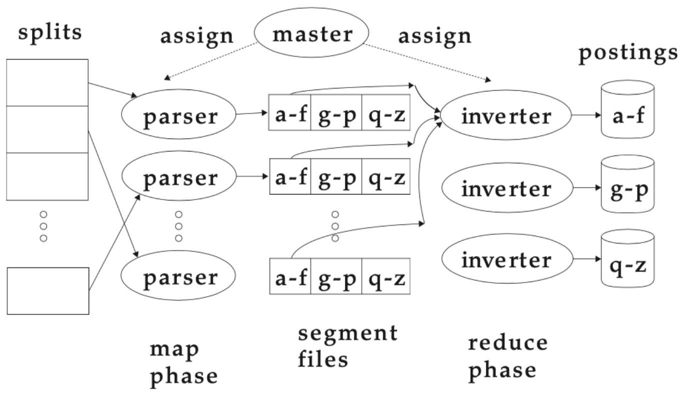
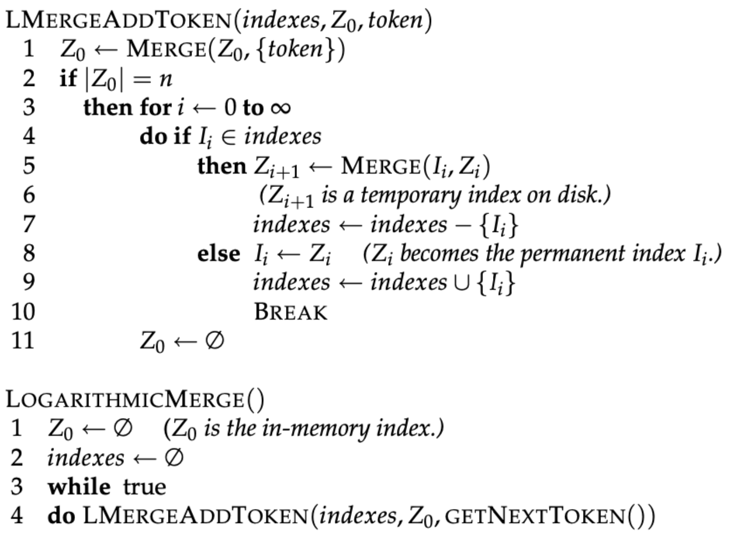


Figure 3: process of distributed index

5 Dynamic Indexing

Up to now, we have assumed that collections are static. But,when documents come in over time and need to be inserted, deleted or modified, it means that the dictionary and postings lists have to be modified. So, the dynamic indexing is introduced here.

In this method, we have to Maintain "big" main index.New docs go into "small" auxiliary index and then, search across both, merge results. Here is the algorithm for the logarithmicmerge for dynamic indexing:



```
LMergeAddToken($indexes, Z_0, token$)
 1  $Z_0 \leftarrow$ Merge($Z_0, \{token\}$)
 2  if $|Z_0| = n$
 3    then for $i \leftarrow 0$ to $\infty$
 4           do if $I_i \in indexes$
 5                then $Z_{i+1} \leftarrow$ Merge($I_i, Z_i$)
 6                       ($Z_{i+1}$ is a temporary index on disk.)
 7                       $indexes \leftarrow indexes - \{I_i\}$
 8                else $I_i \leftarrow Z_i$   ($Z_i$ becomes the permanent index $I_i$.)
 9                       $indexes \leftarrow indexes \cup \{I_i\}$
10                       Break
11           $Z_0 \leftarrow \emptyset$

LogarithmicMerge()
 1  $Z_0 \leftarrow \emptyset$   ($Z_0$ is the in-memory index.)
 2  $indexes \leftarrow \emptyset$
 3  while true
 4  do LMergeAddToken($indexes, Z_0$, getNextToken())
```

Figure 4: algorithm of dynamic indexing

And here is the process of merging indexes:

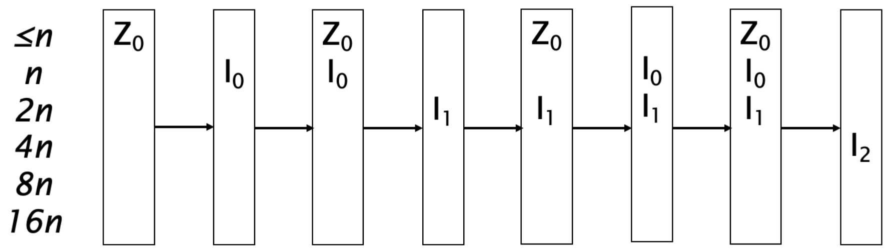


Figure 5: process of logarithmic merge

As you can see in figure 5, we keep the I_max as the main index. If there are two auxiliary index I_k of the same size, merge them into a bigger index $I_(k+1)$. Finally, we could get several indexes including one main index and some or none auxiliary indexes. With these indexes, we search across each one and merge the results to get the final result.