Chapter 4:
Large Scale Indexing

Reading:

- Chapter 4

We now have tolerant retrieval systems for phrase queries, spelling correction, wildcard queries, and general indexing. But what happens when our index gets too large to fit in memory?

Memory requirements of the positional index

Ignoring any overhead, and the size of the vocabulary, and using 4-byte integers for document IDs and positions, how much memory does a positional inverted index require?

- 4 bytes per document containing a term
- 4 bytes per position in that document
- = less space than the entire corpus itself, but can be very large for large corpuses

WHAT HAPPENS WHEN THIS STRUCTURE BECOMES LARGER THAN MAIN MEMORY???

Properties of on-disk data

How is representing data on disk different than in main memory?

- Draw the layout of our positional inverted index in memory. Show:
  - the vocabulary hashmap
  - an array list of Postings objects
  - a Posting object consisting of a doc ID and a list of integer positions
- What is the disk equivalent of a pointer?
- Why can't a similar structure be put on disk?
  - Hardware constraints
  - Files are contiguous
  - Don't be rude re: file system
- Then what do we do? Flatten the pointer structure into contiguous memory.

Index on a disk

So we need to save the index on a disk. Let's assume we can maintain the vocabulary in main memory and only need to move the postings list out of main memory. Then we need to **write** postings lists to disk in a **binary format** (WHY?). We will write a single postings list at a time in the following format

$$\langle df_t, id_{d_1}, tf_{t,d}, p_{t,0}, p_{t,1} \ldots, id_{d_2}, tf_{t,d} \ldots \rangle$$

where

- df_t: the # of documents containing term t
- id: the id of one document containing t
- $\text{tf}_{t,d}$: the number of times t occurs in d

- p_i: the ith position of t in d

These are written to disk in alphabetical order (HOW DO WE GET THIS?). The dictionary then, instead of mapping from a string to a postings list, maps from a string to a BYTE POSITION in this file.

To answer a query:

1. Use the dictionary to retrieve a term's byte position within the postings file.
2. Seek that position on disk.
3. Read the postings from the file into main memory.
4. Construct and return an array/list of the postings
5. Do normal merge routines as before.

SPIMI

If the postings are too large to fit in main memory, then how do we even constuct the index in the first place?

Side trip - disk-based sorting:

If I asked you to sort a list of numbers that is currently on disk, which is too large to all fit in memory at the same time... how would you do it?

Single Pass In Memory Indexing - SPIMI:

1. Open a file for `output`; create a hashtable for the `index`
2. As long as there is free memory:
   (a) Read a token from a document.
   (b) Add the term, document, and position into `index`
3. When memory is exhausted, `sort` the index vocabulary.
4. `write` the `index` to disk to a new file, including the vocabulary terms
5. `clear` the `index`
6. Return to step 2 until all documents are processed.
7. Merge all index files into one final index

To merge:

1. Say we have files $P_1, P_2, \ldots, P_n$ from the indexing step. Open a file-read stream for each.
2. Open a file-write stream for P, the final postings file.
3. Construct a priority queue PQ and add to it the first postings list from each P_i. Associate with each term the stream that the term came from.
4. While the queue is nonempty:
   (a) Pop the first element of PQ into a list L. Repeat until popping some list l_c whose term is not equal to the terms of the elements previously popped.
   (b) Merge the postings lists in L into one final postings list for the term. Write this list to P.
   (c) For each file that had a list with the term just processed, read the next list from that file and place in PQ.
   (d) Clear L, add l_c, repeat until all files are exhausted

We now have 1 index file to use as we wanted!

Vocabulary on disk

We are assuming that the dictionary mapping terms to file offsets fits in main memory. What if it doesn't? NOW we need a way of quickly searching for a particular term in a data file in order to find what byte location its postings list occurs in the index file. How???

Binary search:

Write each string in the vocabulary to one file. Call this `vocab.bin`. We want to search this file quickly. If it is written in SORTED order, we can use a binary search.

Problem: how do we know how long each string is? how do we know where each string begins? how do we "jump" to the middle of the vocabulary list on disk?

Solution:

1. Build a second file, `vocab_table.bin`.
2. For each term, write two values to the table:
   (a) the **byte location** in `vocab.bin` where the term's string is located
   (b) the **byte location** in `index.bin` where the postings list is located.

To satisfy a query q, we now:

1. Jump to the middle of `vocab_table.bin` (HOW?), read the location of `vocab.bin` for the term itself, and compare. Perform binary search routine based on the comparison.
2. When finding an entry in the table for q, read the postings from `index.bin`

Binary tree:

Build a binary tree. Keys: terms. Values: pointers to postings location.

- Can't keep entire tree in memory for same reasons as above, so each node is written to disk. When going left/right in tree, read next node from disk.
- How many reads for an entire vocabulary? Suppose 500,000 tems. A balanced tree will have height $h = \lceil \log_2 (500000) = 19 \rceil$, so up to 19 reads to find a postings location.
- At 5ms per seek, up to 95 ms to FIND the postings location for a term.
- The PROBLEM here is the number of disk reads. Can we do better?

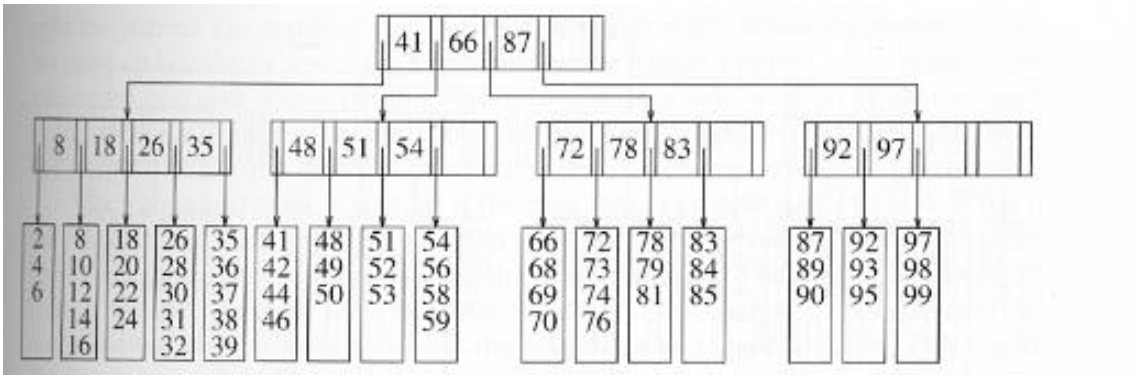
B+ tree:

Operating systems do not actually write and read bytes at a time; they write and read *blocks*. Each read from the disk actually reads an entire block, regardless of how little data the read requested. We can take advantage of this by cramming as much useful information into one block as we can, and buliding a structure around block size.

A **B+ tree of order** m is an m-ary tree with the following properties:

1. All data is stored at leaves.
2. All internal nodes store up to $m - 1$ keys to direct searches into the tree.
3. The root is either a leaf or has 2 to m children.
4. All nonroot internal nodes have between $\lceil \frac{m}{2} \rceil$ and m children.
5. All leaves have the same depth in the tree and have between $\lceil \frac{L}{2} \rceil$ and L children that are **values**, for some L that is proportional to the block size of the secondary memory.

Example 1. What are m and L for this tree?



How does this help our vocabulary?

- Use string keys in the tree.
- In the leaves, associate the string key (the term) with a byte location in the postings file.
- Select m and L based on this.

Example 2. Suppose 16 bytes are needed to save each string in the vocabulary; each pointer to another node in the tree requires 8 bytes; and each value in a leaf stores a 32 byte string with an 8 bytes byte offset. If the block size is 4096 bytes, find m and L so that each leaf and internal node fits into one block.

Each internal node has m pointers to child nodes ($= 8m$ bytes) and $m-1$ keys ($= 16\,(m-1)$ bytes) $= 24m - 16$ bytes. Solving $4096 \geq 24m - 16$ gives $m = 170$.

Each leaf has L value pairs. Each value is 24 bytes. Solving $4096 \geq 24L$ gives $L = 170$.

So we fit 170 terms into each leaf, and the tree is a 170-ary tree. When balanced (guaranteed), the tree is $h = \log_{170}(500000) = 3$ levels deep. 3 disk reads for any string in the vocabulary!

Dynamic Indexing

What to do when the corpus changes?

1. Rebuild the entire thing. When is this acceptable?
2. Collect changes into an *in-memory auxiliary index* Z. What does this mean about satisfying queries?
3. What happens when Z grows too large to fit in memory? Merge it! But merging large indexes is difficult :(.

Linear merge:

Suppose $|Z| = M$, and the number of tokens in the corpus is T. A linear merge strategy is to process postings into Z until memory is exhausted (M tokens), then to merge Z with an on-disk index I, replacing I with the merged result. Each time we generate a new I, we call that a "generation" of I, numbered with a subscript starting at 1.

Assume WLOG that $T = M \cdot c$, that is, some integer multiple of M. Then Z will be filled and then merged with I exactly c times. On the first merge, there is no I and so M postings will be written to disk as I_1. On the second merge, $|I_1| = M$ so $M + M = 2M$ postings have to be merged and written to disk as I_2. On the third merge, we have $2M + M = 3M$ postings written as I_3, etc. This continues until c merges have been performed, giving us a final I_c of size T.

In the end, I_c was constructed by writing cM postings to disk after merging M postings from Z with $(c-1)\,M$ postings from I_{c-1}, which itself was constructed by merging M postings from Z with $(c-2)\,M$ postings from I_{c-2}, which itself was (etc.) down to merging M postings with I_0 which was empty. Since $c = \frac{T}{M}$, the total number of postings written to disk is

$$\begin{aligned}
&cM + (c-1)\,M + (c-2)\,M + (c-3)\,M + \ldots + M \\
=&\sum_{i=1}^{c} i \cdot M \\
=&M \sum_{i=1}^{c} i \\
=&M \cdot \frac{c\,(c+1)}{2} \\
=&\Theta\left(M \cdot c^2\right) \\
=&\Theta\left(\frac{T^2}{M}\right)
\end{aligned}$$

that is, a quadratic-time algorithm in the size of the corpus.

Logarithmic merging:

We can do better if we use multiple on-disk indexes of different sizes. Let $I_1, I_2, \ldots, I_k$ be on-disk indexes of generation $1, 2, \ldots, k$. Let $|I_k| = 2^{k-1}M$, where again $|Z| = M$. Each time Z fills, we check to see if I_1 exists. If it does not, we save Z to disk as I_1. But if I_1 exists, we now have two indexes of generation 1 (Z and I_1); we them to form a new index. Since $|I_1| = |Z| = M$, the new index is of size $2M$. We save this new index as I_2.... unless I_2 already exists, in which case we merge it to make a new index of size $4M$ and save it as I_3.... unless, etc.

Assume WLOG that $T = 2^c M$ for some integer c. When indexing is done, we will have a single index I_{c+1} of generation $c+1$ and size $|I_{c+1}| = 2^{c+1-1}M = 2^c M$. I_{c+1} was created by merging two indexes of generation c, each of size $2^{c-1}M$. The two indexes of generation c were each created by merging two indexes of generation $c-1$, which were created from two indexes of generation $c-2$, and so forth.

Each merge to create generation i requires $2^{i-1}M$ postings to be written to disk. Since $c = \log_2(T/M)$, the total number of postings written during indexing is

$$\begin{aligned}
&2^c M + 2\left(2^{c-1}M + 2\left(2^{c-2}M + \cdots\right)\right) \\
=&\sum_{i=1}^{c+1} 2^{c+1-i} \cdot 2^{i-1} M \\
=&\sum_{i=1}^{c+1} 2^c M \\
=&(c+1) \cdot 2^c M \\
=&\left(\log_2\left(\frac{T}{M}\right) + 1\right) \cdot T \\
=&\Theta\left(T \cdot \log\left(\frac{T}{M}\right)\right)
\end{aligned}$$