Indexing and Sorting for the WWW

We will devote a couple of lectures of the ADS thread to Algorithms for the WWW. The material is more applied than most other topics though all are concerned with appropriate efficient algorithms to various problems.

11.1 Indexing

An *inverted index* to a set of documents (or webpages on the web) is conceptually similar to the index of an ordinary textbook. The idea is that we have a set of *terms* that appear in the text of the documents. Sometimes every individual word appearing in the documents is a term; possibly with some pruning to account for upper case versus lower case letters (*case folding*) and linguistic features[1]. At other times we have a restricted set of given terms. There are other types of index, but this is the most common and is therefore the one we concentrate on here.

An index can be in many forms, but the standard form is that for every term t, there is an associated list of document identifiers $d_1, \ldots, d_{n_t}$, where n_t is the number of documents in which t appears. Therefore once we have the index we can search our set of documents very quickly by finding the term t in our index (probably using binary search). In Figure 11.1, we show a "set of documents" each comprising a single line of a particular nursery rhyme. In Figure 11.2 we show two inverted indexes for this "set of documents." The figures are taken from Chapter 3 of the *Managing Gigabytes* book mentioned at the end of these notes.

Document	Text
1	Pease porridge hot, pease porridge cold,
2	Pease porridge in the pot,
3	Nine days old.
4	Some like it hot, some like it cold,
5	Some like it in the pot,
6	Nine days old.

Figure 11.1. A children's rhyme, each line being treated as a document .

Each of the two indexes of Figure 11.2 consist of a *lexicon* (the set of *terms*), and for every term in the lexicon, an inverted file entry for that term. An inverted file entry is the frequency of a term followed by a list of locations where that term appears in the document set; for example, in the left index, the inverted file for the term 'hot' is $\langle 2; 1, 4\rangle$. Note that both of the Indexes has a frequency of 2 (appears in two documents) for every term in the lexicon—this is just a coincidence for this particular example.

[1] If we are indexing documents in English it makes sense to regard *words* and *wording* as occurrences of the same stem: *word*. This is referred to as *stemming*.

Number	Term	Documents
1	cold	$\langle 2; 1, 4\rangle$
2	days	$\langle 2; 3, 6\rangle$
3	hot	$\langle 2; 1, 4\rangle$
4	in	$\langle 2; 2, 5\rangle$
5	it	$\langle 2; 4, 5\rangle$
6	like	$\langle 2; 4, 5\rangle$
7	nine	$\langle 2; 3, 6\rangle$
8	old	$\langle 2; 3, 6\rangle$
9	pease	$\langle 2; 1, 2\rangle$
10	porridge	$\langle 2; 1, 2\rangle$
11	pot	$\langle 2; 2, 5\rangle$
12	some	$\langle 2; 4, 5\rangle$
13	the	$\langle 2; 2, 5\rangle$

Number	Term	Documents;Words
1	cold	$\langle 2; (1; 6), (4; 8)\rangle$
2	days	$\langle 2; (3; 2), (6; 2)\rangle$
3	hot	$\langle 2; (1; 3), (4; 4)\rangle$
4	in	$\langle 2; (2; 3), (5; 4)\rangle$
5	it	$\langle 2; (4; 3, 7), (5; 3)\rangle$
6	like	$\langle 2; (4; 2, 6), (5; 2)\rangle$
7	nine	$\langle 2; (3; 1), (6; 1)\rangle$
8	old	$\langle 2; (3; 3), (6; 3)\rangle$
9	pease	$\langle 2; (1; 1, 4), (2; 1)\rangle$
10	porridge	$\langle 2; (1; 2, 5), (2; 2)\rangle$
11	pot	$\langle 2; (2; 5), (5; 6)\rangle$
12	some	$\langle 2; (4; 1, 5), (5; 1)\rangle$
13	the	$\langle 2; (2; 4), (5; 5)\rangle$

Figure 11.2. Two indexes for the "set of documents" of Figure 11.1. The left index gives frequency and list of document numbers. The right index gives frequency, document numbers and occurrence within the relevant document.

The *granularity* of an *inverted index* is the detail to which we record the location of a term in the document. In the index on the left of Figure 11.2, the granularity is document-level. In the index on the right of that Figure, the granularity is word-level.

The terms of a lexicon have an associated ordering, given by the ordering of their term-numbers. For the indexes in Figure 11.2 the ordering is alphabetical, however this is not generally the case (and the mapping of terms to term numbers will need to be stored).

11.2 Querying using an Index

An Inverted Index enables fast search on a set of documents, when queries are formulated using the terms of the lexicon. The quality of the query replies will partly depend on the granularity of the *Inverted Index*: for example, if we perform the search "pease" on the left Index of Figure 11.2, then we are informed of the identifiers of the two documents which contain "pease", but we do not know where the term is located in the document (obviously not a big problem with the 'documents' of Figure 11.1). We would need to check the document itself to look for the position of that term. However, with the finer right-hand Index in Figure 11.2, we will directly recover the positions of the terms in the documents.

The queries we might need to make are usually more complicated than single-term queries. However, assuming that our Inverted Index stores the inverted file entry for every term t in order of document index, then we could use a *merging* technique to answer simple boolean queries such as "pease" AND "hot". Solutions to boolean queries can be generated in time linear in the length of lists for the two terms, by performing a synchronised scan of the two lists in a manner similar to the merge routine of mergeSort. The operation performed by this scan will be

$\cap$ (intersection) in the case of AND and $\cup$ (union) in the case of OR, and can be applied inductively, still in linear time.

In practice, when our documents are webpages, we will want to rank the pages, rather than treat them all as equal. We will discuss this issue (in the context of the PageRank algorithm of Google) in a subsequent lecture.

11.3 Constructing a Large-Scale Index

Indexing is the act of preprocessing a set of documents to construct an inverted index for those documents. It is this process that we are concerned with in this lecture. We are concerned with "indexing in the large" where the set of documents to be indexed might take up many gigabytes of memory. Therefore in addition to considering asymptotic running times of our algorithms, the hidden constant inside these terms will be of interest to us. Also, we will be concerned with the amount of memory used by our algorithms, as well as the amount of disk space used.

If we set aside some concerns there are various efficient algorithms that could be used to construct an inverted index. For example, if we are not concerned about the amount of memory available (which up to now has always been the case), we could just extract all $\langle t, d\rangle$ pairs from all documents and then sort these terms in memory. In Section 11.4 we show how to do something like this in a bit more detail. However, the scenario that we consider in this Lecture Note (and which arises very frequently in indexing for the web) is one where the computer performing the indexing is being stretched to its very limit. Hence the goal is to get the computer to handle as many documents as possible while leaving enough space to perform computations on those documents. We are happy to work partly from the disk (rather than in memory) in doing the indexing if this will allow more documents to be handled. On the other hand disk-accesses must be rare (since moving to a new location on disk is very expensive compared to main memory).

11.4 Memory-Based Inversion

If we do not need to worry about the amount of memory available, then we have a simple algorithm. We can take each document in turn, parse it, and do lexical analysis on that document (removing stop words like "the"), to extract the terms for the document in the form of $\langle t, d, f_{d,t}\rangle$ triples (where d is the document, t a term, and $f_{t,d}$ the frequency of that term in the document). Then we only need to take the union of the document lists and sort them. One way we can choose to implement this is as a *Dictionary* implementation where the *items* we store in the dictionary are *(key, list)* pairs, where each list is a list of $\langle d, f_{d,t}\rangle$ entries (for its associated term t). On recovering a set of terms from a new document d, we only need to search the dictionary and append relevant $\langle d, f_{d,t}\rangle$ pairs to the list attached to key t (inserting a new item if necessary). This is the approach we take in Algorithm 11.3.

The *Dictionary* data structure can be any *Dictionary* structure, but for efficiency it is better if it is a structure that keeps the elements somehow in sorted

order. AVL trees give $\Theta(\lg n)$ performance for inserting, searching and deleting; however hash tables (which don't have smaller overheads) can be more efficient in practice.

Algorithm memoryBasedInversion(D)

1. Create a *Dictionary* data structure.
2. **for** $i \leftarrow 1$ **downto** $|D|$ **do**
3. Take document $d_i \in D$ and parse it into index terms.
4. **for** each index term t in D_i **do**
5. Let $f_{d_i,t}$ be the frequency of t in d_i.
6. If t is not in S, insert it.
7. Append $\langle d_i, f_{d_i,t}\rangle$ to t's list in S.
8. **for** each term $1 \leq t \leq T$ **do**
9. Make a new entry in the *inverted file*.
10. **for** each $\langle d, f_{d,t}\rangle$ in t's list in S **do**
11. Append $\langle d, f_{d,t}\rangle$ to t's *inverted file entry*.
12. Append t's entry to the *inverted file*.

Algorithm 11.3

Sometimes we might compress t's inverted file entry before we save it to the inverted file in line 12.

Observe that with this Algorithm, the *term numbers* that label the terms can be considered to be in the same order as the terms themselves (as in Figure 11.2). We do not mention this explicitly, but it is implicit in our storing of data in S according to the key t.

The inversion time $T_I(D)$ required by this algorithm consists of

- The time $T_p(D)$ to read, parse and lexically analyse all the documents (reading and parsing takes linear time with a small constant, and if we are lucky, the same will be true for lexical analysis).
- The time $T_q(D)$ to query S and append an entry to an item of S for every $\langle t, d\rangle$ pair in D (this is $O(n \lg(n))$, where n is the number of $\langle t, d\rangle$ terms in D) .
- The time $T_w(D)$ to implement the loop in lines 8-12 to write the *inverted file*, linear in the size of the Inverted Index (the number of $\langle t, d\rangle$ terms in D).

This is a simple and reasonably-efficient algorithm. Due to the removal of stop words and recurrences of the same term in a given document, it is very likely that $O(n \lg(n))$ is no larger than a small multiple of D (and in any case $O(n \lg(n))$ is only an upper bound). This indexing algorithm probably takes linear time in the size of the input.

However, this is not our main concern. The efficiency of an Indexing algorithm is measured in hours and not in asymptotic notation. In *Managing Gigabytes* some actual figures are given for this algorithm. Another issue is the limit that exists on the size of the set of documents, due to the limited size of the memory. This raises the question of whether we need to work 'in memory' all the time. Could we store some of the Documents (or the partially constructed index) on disk throughout the algorithm (we might want to store the inverted file to disk after the algorithm has finished)? The issue for disk access is moving a head that reads from the disk; however, sequential access of a large block of the disk will not take much more time than the original access. It is true that we could implement lines 1-7 on disk without loss of performance. That is because the entries are added into the data structure in a dynamic fashion; when allocating a new vertex (if we are using an AVL implementation of *Dictionary*), it is allocated sequentially in memory (even if we think of it in pointer form). However, for lines 8-12, it is not possible to work with part of S stored in memory. When we examine the linked list of $\langle d, f_{d,t} \rangle$ entries for term t (in order to append them together onto the *inverted file*), we cannot expect these to be close to each other in memory (or on disk, should we take this approach). This phase of the algorithm would be incredibly slow on disk (again, the numbers involved are discussed in *Managing Gigabytes*).

11.5 Sort-Based Inversion

We now present an alternative algorithm for constructing an inverted index that does allow the disk to be used intelligently throughout the algorithm. The *temp file* used in the algorithm refers to a file on disk. This algorithm uses mergeSort to sort files that lie on disk; mergeSort can be used as an *external sorting* algorithm, because it processes its input in a sequential fashion).

External Merge Sort

In the sorting context, External MergeSort is used when we want to sort a set of n items, and where $n \gg K$) (read $\gg$ as "much bigger than"), where K is the number of items that we can fit into the computer's memory.

Before going into details it is worth noting External MergeSort is really a very simple idea. The key intuition of MergeSort (internal or external) is that if we merge two sorted sequences together we get a new single sorted sequence. So if we make sure we have sorted blocks of whatever size in an input file we merge them in pairs (possibly with one left over) and keep doing this till there is just one block. So if we could pull in complete sorted blocks from the input file and merge them together all in main memory we could then write out the new merged block to a secondary file. Keep doing this till all blocks have been processed. We could then copy the auxiliary file back to the input file but that would be a waste fo time, we can just treat the auxiliary file as our new input file and the previous input file as the auxiliary file. However if we are sorting a huge amount of data there comes a point when we cannot hold entire sorted sequences in main memory, only a part. The key insight here is to note that in order to

merge two sequences all we need to know are the current (as yet unprocessed) elements of each sequence. We compare these and then get the next element from the relevant block. For the merged result we could if we wanted just write it out to file immediately. So in principle, we could just always fetch one element from each input block, do a comparison and write out the smallest element to the output (auxiliary) file. But this is a very bad idea because each access to an external file is very expensive. It is much cheaper to ask for a sequence of consecutive items to be fetched from the external disk. Since we are limited in main memory we have no option but to allocate a certain amount for each of the two sequences to be merged and a certain amount for the merged result. When we have used up either of the fetched things form the sorted sequences we ask for another lot till there is no more from that sequence. When we fill up the memory allocated to the merged sequence we flush it out to the output (auxiliary) disk. As to how much of the available memory we allocate to each of these, we are free to choose any policy but it seems reasonable to allocate a third to each (if we know more about our technology we might choose to change this but the principle is the same). Finding exact expressions for the number of iterations is a little tricky and needn't concern you too much. It is the overall principle that you need to understand.

Let's now turn to a more detailed description of the idea with pseudocode given in Algorithm 11.5. The n items to be sorted are stored on disk-A (analagous to array A in standard mergeSort) and will be sorted into disk-B (analagous to the scratch array B in standard mergeSort). We initially break the data on disk-A into n/K sequentially arranged "blocks", each of size K, and individually sort each of these blocks in memory[2]. Then External MergeSort performs a "bottom-up" version of mergeSort (note that disk-A and disk-B swap roles of input/output disk as j is incremented during externalMergeSort).

Note that during lines 5-12, externalMergeSort needs to access consecutive blocks of the current input disk. Once a the start of a block is found all subsequent accesses are sequential. Moreover once we have finished with a pair of blocks the pointer to the second block is now at the start of the first block of the next pair. Hence the number of non-sequential disk accesses is approximately $\sum_{j=1}^{\lceil \lg(n/K) \rceil} (n/(2^j K) + 1)$, which is $\Theta(n/K)$. Aside from this there is a pointer to the current output-disk.

Finally note that there are variations on this algorithm but the differences are in the details (e.g., what buffering is used) rather than the overall idea.

Sort-based Inversion

In contrast to Algorithm 11.3, Algorithm 11.5 does not output the terms in alphabetical order. Instead they are output in the order in which they have their "first appearance" in the set of documents. This is a result of the fact that we will perform the processing of the documents hand-in-hand with the construction of the index; hence we need to construct parts of the index before we know all the

[2]For the sake of simplicity we assume that in expressions such as n/K we obtain an integer. Taking care of cases when this is not so would just obscure the idea.

Algorithm externalMergeSort(A)

1. **for** $i = 1$ **to** n/K **do**
2. read block-i of disk-A (containing K items) into memory;
3. sort block-i in memory using any 'in-place' algorithm (eg quicksort);
4. write the sort of block i out to disk-B.
5. /* disk-B now becomes current input-disk */
6. **for** $j = 1$ **to** $\lceil \lg(n/K) \rceil$ **do**
7. **for** $i = 1$ **to** $(n/2^j K)$ **do**
8. buffer the first $K/3$ entries of block-i and block-$i+1$ from *current input-disk* into memory ;
9. initialize the output buffer b (of size $K/3$);
10. **while** there are items left to sort **do**
11. perform externalMerge on the small blocks in-memory
12. /* outputting buffer b when it is full, and inputting
13. more of block-i/block-$i+1$ when needed */
14. **od**
15. swap role of *current input-disk* between A and B.

Algorithm 11.4

terms that will belong to the final index. The ordering is memoized by associating with every term t a *term number* τ, which marks its position in the ordering.

Throughout the presentation of this algorithm, K denotes the number of inverted file entries $\langle t, d, f_{d,t} \rangle$ that can be held in memory (clearly this will not be a tight fit, rather K will be an upper bound set to ensure that the algorithm has enough working memory under these circumstances).

Algorithm sortBasedInversion(D)

1. Create a *Dictionary* data structure.
2. Create an empty *temp file* on disk.
3. **for** $i \leftarrow 1$ **downto** $|D|$ **do**
4. Take document $d_i \in D$ and parse it into index terms (etc).
5. **for** each index term t in D_i **do**
6. Let $f_{d_i,t}$ be the frequency of t in d_i.
7. Check whether $t \in S$ (and check term number τ).
8. If $t \notin S$, insert it (with the next free term number τ).
9. Write $\langle \tau, d_i, f_{d_i,\tau} \rangle$ to *temp file* (τ is t's term number).
10. Call externalMergeSort on *temp file*, to sort in order of $\langle \tau, d \rangle$ (with memory size K);
11. /* *temp file* now sorted. Output inverted file. */
12. **for** $1 \leq \tau \leq T$ **do**
13. Start a new *inverted file entry* for t (term number τ).
14. Read the triples $\langle \tau, d, f_{d,\tau} \rangle$ from *temp file* into t's entry.
15. Append t's entry to the *inverted file*.

Algorithm 11.5

The main difference between our new algorithm and Algorithm 11.3 is the way we perform sorting. In our memory-based algorithm we never *directly* performed any sorting. However by working with a (say) AVL tree implementation of *Dictionary*, and inserting new terms into that, and by appending new $\langle d, f_{d,t} \rangle$ entries to the end of the list for t, the effect was to perform insertionSort on the $\langle t, d \rangle$ pairs[3] in our set of documents. However, as we discussed in Section 11.4, Algorithm 11.3 cannot be adapted to use disk space. In Algorithm 11.5, we make use of externalMerge.

Our Sort-Based algorithm is organised in a few phases. At the beginning of the algorithm (lines 1-2), we set up a *Dictionary* (which will just contain (*term*,

[3] In fact, it is a good time to mention that in general, using a balanced tree (either an AVL tree or a red-black tree) is an immediate way of modifying insertionSort so that it runs in $\Theta(n \lg n)$ time in the worst-case; recall that insertionSort has worst-case time $\Omega(n^2)$ in its standard linked list form.

term number) pairs) and we open a disk file called *temp file*. In the first phase of the algorithm (lines 3-9), we examine the documents in sequential order, and append all $\langle \tau, d, f_{d,\tau} \rangle$ triples (in sequential order) into our disk file (updating the lexicon whenever we find a new term). Next in line 10 we call externalMergeSort to sort the entire *temp file* in order of $\langle \tau, d \rangle$. Recall from our discussion about externalMergeSort that this uses $\Theta(n/K)$ (specifically, about $3(n/K)$) disk accesses; since these are very expensive, we need a computer with a sufficiently large value of K (so we don't need too many runs). Finally, in lines 11-15, we use the sorted *temp file* to construct the Inverted Index on disk.

There are some issues for Algorithm 11.5 that were not relevant for Algorithm 11.3 but the most important one is disk accessing. For Sort-Based Indexing, we will require K to be smaller than n by some constant factor c (otherwise we are not saving any space in using the disk). However, if c is too large, the disk accesses begin to downgrade the running-time of the algorithm. Hence we need to keep a sensible balance (see *Managing Gigabytes* for numbers).

As with Algorithm 11.3, we do have the option of compressing an inverted file entry before appending it into the inverted file index. We should also note that it is even possible to work with a compressed format of the temporary file throughout the execution of Algorithm 11.5, leading to optimized performance. The compression has the effect of shrinking the size of the files stored on disk (or in memory), and therefore reduces the number of disk accesses that need to be performed by the algorithm. The merge operation can still be performed on the runs stored on disk, as long as care is taken. Hence with compression we can obtain a further speeded version of our Sort-based algorithm.

11.6 Further Reading

Neither [GT] nor [CLRS] present any material on Algorithms for the WWW.

A good textbook is *Managing Gigabytes*, by Ian. H. Witten, Alistair Moffat, and Timothy. C. Bell. The relevant chapter for this lecture is Chapter 5 (and some parts of Chapter 3). This book gives numbers for the time and space taken by various Index-constructing algorithms (in terms of hours, gigabytes etc).

There is also a lot of information available online. For example, here are two papers (both are rather application-specific):

- Building a distributed Full-test Index for the Web, by S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. *ACM Transactions on Information Systems (TOIS)*, **19**(3). Online at:
  `http://www10.org/cdrom/papers/275/`

- Very Large Scale Information Retrieval, by David Hawking. In *Text and Speech Triggered Information Access*, Eds. Gregory Grefenstette and Steve Renals, 2003. Online at:
  `http://www.inf.ed.ac.uk/teaching/courses/tts/papers/hawking.pdf`

Exercise

Simulate the externalMergeSort algorithm of this note. We assume that $K = 6$ and the input data (held on Disk A) consists of:

$$25, 2, 48, 48, 2, 36, 36, 9, 25, 9, 26, 33, 7, 46, 1, 8, 20, 40, 38, 3, 43, 12, 18, 9$$

After the initial sort of blocks of size K Disk B now has the data in the form:

$$[2, 2, 25, 36, 48, 48], [9, 9, 25, 26, 33, 36], [1, 7, 8, 20, 40, 46], [3, 9, 12, 18, 38, 43]$$

where we have delineated each sorted block by putting it in square brackets (this is just for convenience). Proceed to simulate the algorithm, a good level of granularity to use is to show what happens at each read or write to disk of data blocks of size $K/3$. As each data item is processed you could cross it out or underline it. Here is a possible diagram:

Disk A :

Disk B : [2,2,25,36,48,48],[9,9,25,26,33,36],[1,7,8,20,40,46],[3,9,12,18,38,43]

RAM:

2			...
2			...
9			...
9			...
			...
			...

The table for the RAM section shows available memory in a column. Each column is used to keep track of the state of the RAM as blocks are read in or written out (it just saves a lot of rubbing out). Each block of size $K/3$ of the RAM is delineated with a double line (of course there is just to help keep track of the simulation). the diagram shows the situation just after sub-blocks of size $K/3$ have been read into the RAM. The next phase is to start merging these.

A complete simulation would be rather long. A good compromise is to simulate to completion the processing of the first two sorted blocks of Disk B as shown in the diagram. Then start the process for the next two blocks but jump to its completion. After that Disk A will have two sorted blocks [what size will they be?]. Do a few steps of the algorithm on those till you are happy that you have understood the process.