Data Structures

We have to distinguish between the abstract model (used for the complexity of algorithms), where everything is in the main memory, and the "real" model: when we have large data structures we may also need an external storage.

External Storage

We cannot assume that each step has the same cost (this is the hypothesis under the complexity analysis of algorithms, reasonable when all the data fit in main memory) when we have to access the external storage, this is even more true when we consider data distributed in a network. Accessing external storage can have costs which can be orders of magnitude greater than accessing the main memory. Fetch from main memory is approximately 100 ns, fetch from new disk location is 8000000 ns.

So, when counting complexity, we should count the number of disk accesses. A few consequences of this choice is that we may want to transfer data in bulks. Since each read is costly, we want to read more than strictly necessary.

We want to do a finer analysis, not only asymptotic results (e.g. the base of the logarithm in the complexity is important). And we want to spend more time in memory in order to reduce the number of accesses (e.g. compress and decompress data).

Data Structures for Dictionaries

Mainly two choices in data structures for dictionary: **hash tables** and **trees**. Indeed linear scanning a large dictionary is usually not very effective since dictionary can be quite large (linear time in number of terms, while we can use logarithmic time in the case of binary search). The choice of the data structure is governed by the number of keys we will be likely to have, the fact the this number will be static or changing a lot, the frequency in which we access different keys.

Hash Functions

A *hash table* is composed by a table of fixed size, usually contiguous in memory, and a *hash function* that gives for each element a position where to store it, not necessarily in any particular order. Sometimes we may have collisions, i.e. the hash function will give the same position to different terms.

A traditional **hash function** is the modulus operator with modulus m size of the table, in case we work with integers, when we have strings it can be a little bit more complicated: we can sum all the numerical values corresponding to all the characters in the string (*component sum*), eventually putting it inside a traditional hash function for integers, or in a more effective way we can see each element of the string as a coefficient of a polynomial of degree $length - 1$

(*polynomial accumulation*), and we evaluate the polynomial for a fixed value of the unknown and again plug the result in a hash function for integers.

So we have that an hash function gives for each term an integer, which is the position inside the hash table, the drawback is that we can have **collisions** (i.e. the hash function returns for two different input terms the same value). The number of collisions is approximately computed by the **Load facotr**: $\frac{\#elements}{sizeofthetable}$, but still we can have collisions even if the table is not already full. In order to lower the load factor we can use more memory, with higher load factor we have lower memory usage but higher risk of collisions. Hence:

- lower load factor $\rightarrow$ higher memory usage but less risk of collisions;
- higher load factor $\rightarrow$ lower memory usage but higher risk of collisions.

We cannot remove terms that have been already inserted in a table, we have to manage collisions. One way is **open addressing**: all entries are stored in the table, in case of collision the first free slot according to some probe sequence is found. Another way is **chaining**: each cell is a list of all entries with the same hash. Or **perfect hashing**: for a fixed set it is possible to compute an hashing function with no collisions, or other collision resolution techniques (like **cuckoo hashing**).

Finding an element in a hash table requires $O(1)$ *expected* time, in some cases this is also the worst case e.g. in perfect hashing. Adding new elements might require *rehashing* (reinsertion of all elements in a bigger table) which is costly, but done in order to keep the load factor low enough. The main drawback is that some kind of searches are not possible (in general anything that require something different than the exact term), like looking for a prefix, or for a wildcards (all word containing a substring).

Binary Trees

They are trees in which each node has at most two children, each node has an associated value (in our case it will be a term). A **binary search tree** has the property that, if we have a total ordering (in case of terms we use the lexicographical order) of the tree, then all the elements in the left subtree are smaller then the value in the root, while the ones in the right subtree are larger. If a tree is balanced, then the search can happen in $O(logn)$ steps (we assume not to have duplicates).

Binary trees have a lot of advantages over hash tables: insertion and deletion are not expensive (even keeping the tree balanced), we can search for a prefix (since everything is ordered, we can reduce the search looking only in subtrees). However the main drawback is that, if our tree is not stored in main memory, each step requires an access to external storage that might be costly.

In order to reduce the problem of accessing too often the main memory, we can use **B-tree**.

B-trees

B-trees are a generalisation of binary search trees, in which each node has between a and b children, and pointers between them. We need to select the number of elements in a node, when we access the main memory we have the block size, which is the minimum amount of data that the disk can read (usually 512bites for disks), so we select the size of our node so that it fits exactly into this block (B is the size of the block), so we reduce the number of disk accesses. The number of accesses in BST was $\log_2 n$, now it is $\log_{\#elementspernode} n$. This structure is used often for filesystems.

A B-tree may be viewed as collapsing multiple levels of the binary tree into one.

Remark that $\lceil log_B(\#terms) \rceil$ corresponds to the number of disk accesses, which are the ones dominating the running time.

Tries

The best candidates for now for representing dictionaries are B-trees, but we can also use data structures that are specific for strings (or sequence of symbols), such are **tries** (also called prefix trees/arrays). The idea of the trie is to build a special kind of tree where simply looking at the element of the key gives us the pass inside the tree: we are looking for prefixes. A trie is a special kind of tree based on the idea of searching by looking at the prefix of a key. The key itself (in our case the term) provides the path along the edge of the trie. The advantage of using tries is that the access time depends only on the length of the key, in our case the term, usually we cannot go below this value, since we have to read the term we are searching. So access time is $O(m)$, where m is the size of the key. Insertion is still possible and efficient.

The idea is that to read a term we need to read the path that reaches the term. terms are encoded in the paths from the root of the tree to a node. We are actually splitting anytime we have two terms that start to differ. Now the search is not dependent on the size of the dictionary but on the length of term.

In all our previous analysis we have considered constant length keys. Tries have access time that is as good as the one of the hash table, but here there cannot be collisions. Insertion is efficient (we look at the path and split where necessary). Moreover search inside a range of key is very efficient. We may still have to access the disk, but this is manageable, the idea is that we can simply pack together multiple nodes and read them in blocks, or there are variants of tries to mitigate the access time problem.

Dictionary and Index Compression

There are a number of compression techniques for dictionary and inverted index that are essential for efficient IR systems. The major benefit of compression is clear. we need less disk space. Another advantage (also in addition to caching) is faster transfer of data from disk to memory. So, in most cases, the retrieval system runs faster on compressed posting lists that on uncompressed ones.

We can compress both the dictionary and the postings (the index is composed of this two things), in order to save disk space, to keep the entire dictionary in memory, or at least more data in memory. It might be faster to read less data from disk and decompress it in memory than to read the non-compressed data.

Usually the size of the dictionary is $M = k \cdot T^b$, being T the total number of tokens in the collection, k is typically between 30 and 100 and usually $b \approx 0.5$. this is called **Heap's law** (given the number of tokens in the corpus we can estimate the size of the dictionary without looking further). We can deduce that the size of the dictionary (also without estimating it) increases with the size of the collection (rather than maximum vocabulary size being reached). Moreover we can deduce that the size of the dictionary is quite large for large collections. This highlights the importance of compression for an effective information retrieval system.

In other words, the Heap's law estimates vocabulary size as a function of collection size. The motivation for Heap's law is that the simplest possible relationship between collection size and vocabulary size is linear in log-log space and the assumption of linearity is usually born out in practice. The parameter k is quite variable because vocabulary growth depends a lot on the nature of the collection and how it is processed (e.g. case folding and stemming reduce the vocabulary size, including numbers and spelling errors increase it).

Now we also want to understand how terms are distributed across documents. This helps us to characterize the properties of the algorithms for compressing posting lists.

Heap's law connects also with the **Zipf's law** (which is a commonly used model of the distribution of terms in a collection), according to which the i-th most common term in a collection appears with a frequency $cf_i \propto \frac{1}{i}$, meaning that the frequency of terms decreases rapidly with the rank (the distribution of terms in the corpus is not uniform across documents). So most of the terms will be present in a very small number of documents, very few will be present across all the corpus.

Recall that one of the primary factors in determining the response time of an IR system is the number of disk seeks necessary to process a query. Thus, the main goal of compressing the dictionary is to fit it in main memory, or at leats large portion of it, to support high query throughput.

We can consider **dictionary as fixed-width** (m) **entries** with a pointer to the posting list (this is true in all data structures except tries): in this model words of at most m characters can be stores, and we waste a lot of space for short words. Using fixed-width entries for terms is clearly wasteful. We can overcome the shortcomings of this structure by storing the dictionary terms as one long string of characters.

The second possibility is to keep all the terms in a contiguous piece of memory (**dictionary as single string**), one after the other, and store inside the data structure used two pointers: one that points to the first character of the term, and one pointing to the posting list. In this way we are not waisting space to store words (only characters actually needed for the term), to see where the word ends we can look at the next pointer, but still we have to keep an additional pointer. When we have to update the index, this structure is not so flexible.

We can use **blocked storage**, in which we divide the set of consecutive terms in blocks, having one pointer for each block (and also keep track of the length of the terms), so that we have a pointer every k elements, we can find the terms by linear scanning the block (terms are consecutive and we have the length of each one). It is a tradeoff between space and access time: we can reduce the amount of space used, but then we increase the time that we need to search the specific term in the block. Storing the length of the string as an additional byte at the beginning of the term. We thus eliminate $k-1$ term pointers, but need an additional k bytes for storing the length of each term. By increasing the block size k, we get a better compression, but there is a tradeoff between compression and the speed of term lookup.

Moreover we are still not taking into account the fact that words in the dictionary are ordered alphabetically (it would be useful to consider this fact to compress dictionary).

We can use blocked storage plus **front coding**: we don't store the entire term but only the part that is different from the prefix of the previous term (i.e. a common prefix is identified for a subsequence of the term list and then referred to with a special character), along with the length of the prefix shared with the previous word (and the total length of the term). By using the fact that words are stored in alphabetical order we can drastically reduce the amount of storage needed. This is a tradeoff because in order to search for a term we need to reconstruct (deconding phase in addition to linear scanning) the term using the previous prefix: tradeoff between doing less stuff but in memory or more stuff but accessing the disk, i.e. a tradeoff between reducing size and incrementing the cost of retrieving a term.

When we store posting lists we usually store sets of DocID, but these can be large if we have a very large corpus. However we can use the fact that the list is ordered, we can store instead of the DocId, the difference between the current DocID and the previous (we encode gaps instead of DocIDs, i.e. how big is the jump between one DocID and the next, using the key idea that the gaps between postings are short). In order to recover the DocID we have to sum all the terms before it in the list, so to recover the DocID of the element in position k we need to: $a_0 + \sum_{i=0}^{k}(a_i - a_{i-1})$.

But still we have to store a number for each DocID, and most of the gaps are small, so we can use **variable byte codes** (i.e. encode numbers in a variable amount of bytes) to use less storage. In this way recovering a DocID is more complex, and, most importantly, access to the list must be sequential: to recover a DocID we need to read all the previous ones (but this is the case for union and intersection).

We know that 1 byte is 8 bites, so we can encode $2^8 = 256$ different values in a byte, so we can encode $2^7 = 128$ different values in the first 7 bits and the last bit is a *continuation bit*: if it is 0 then we have completed reading the number, otherwise we must continue to read the next byte. The tradeoff is that we have one bite for each byte dedicated to say if we have finished the reading or not, but most frequent terms will have small gaps in their posting lists (we expect to have a lot of small gaps). So we can store the size of most gaps in only a few bytes.

Updating the Index

We need to organize posting lists on disk: we can keep one file per posting list, but this can lead to too many files for a filesystem to be managed efficiently, or we can have a single large file containing all the postings (i.e. a concatenation of all posting lists, better than one file per posting) or (which is usually done) a combination of the two.

Dynamic indexing

We need to insert and delete documents: we can rebuild the index from scratch, but this is not very efficient, only useful when the number of changes is small. In this way, if we want to keep the system online while reindexing we need to keep both the new and the old file during the change, until the new one is ready (very expensive in terms of memory). This is a good solution if the number of changes over time is small and a delay in making new documents searchable is acceptable (and enough resources are available to construct a new index while the old one is still available for querying).

An idea could be to use **auxiliary index** (especially if we have a requirement that new documents should be included quickly): we keep both the *main index* and the *auxiliary index* (containing only new documents) and a **invalidation bit vector** (in which we save which documents have been deleted, one bit for each document). In order to perform a query, we query both indexes at the same time, we obtain to collections of documents and we merge them, then we filter the result using the invalidation bit vector.

The problem is that the auxiliary index can become too big, and we have to marge it with the main index (the cost of this merging operation depends on how we store the index in the filse system). One thing that we can do is to actually have a collection of indexes, each one of size double of the previous one, $n, 2n, 4n, ...$: the smaller is kept in memory and copied in the disk when full and a new one of size n is created in main memory (empty). When this new index is full, since there is already an index of size n in memory, it is saved on disk and merged with it to form an index of size $2n$. A new (empty) index of size n

is created in memory. When full it is copied to the disk (no merge is necessary, since we have a free slot) and a new slot of size n is created.

This is called **logarithmic merging**. As in the auxiliary index scheme, we still need to merge very large indexes occasionally (which slows down the search system during the merge), but this happens less frequently and the indexes involved in a merge on average are smaller.