1

Searching Strings by Prefix

This problem is experiencing renewed interest in the algorithmic community because of new applications spurring from Web search-engines. Think to the *auto-completion* feature currently offered by Google, Bing and Yahoo in their search bars. That is a prefix search which is executed on-the-fly over millions of strings and uses as pattern the query typed by the user. The dictionary typically consists of the most recent and the most frequent queries issued by other users. This problem is made challenging by the size of the dictionary and by the time constraints imposed by the patience of the users. In this chapter we will describe many different solutions to this problem of increasing sophistication and efficiency both in time, space and I/O complexities.

The prefix-search problem. *Given a dictionary $\mathcal{D}$ consisting of n strings of total length L, drawn from an alphabet of size σ, the problem consists of preprocessing $\mathcal{D}$ in order to retrieve (or just count) the strings of $\mathcal{D}$ that have P as a prefix.*

We mention that other typical string queries are the *exact* search and the *substring* search; the latter consists of finding all *positions* where P occurs within the strings of $\mathcal{D}$. It goes without saying that the prefix search is more general that the exact search because we are not confined to search for the entire string P. So the data structures introduced in this chapter can be used to solve the exact-search problem too. But we have to say that hashing is the preferred solution in practice because of its time/space efficiency. As far as substring search is concerned, Chapter **??** will show that even this sophisticated problem can be *reduced to* prefix search and then it can be solved via the Suffix Array data structure. The present chapter, in the last Section 1.6, will also briefly mention about this reduction and it will sketch the use of String B-trees in place of Suffix arrays. The net result is that the solutions detailed in this chapter are the backbone of many other (practical) solutions whose application goes far beyond the ones discussed below and referring just to prefix searches.

1.1 Array of string pointers

We start with a simple, common solution to the prefix-search problem which consists of an array of pointers to strings stored in arbitrary locations on disk. Let us call $A[1,n]$ the array of pointers, which are *indirectly* sorted according to the strings pointed to by its entries. We assume that each

pointer takes w bytes of memory, typically 4 bytes (32 bits) or 8 bytes (64 bits). Several other representations of pointers are possible, as e.g. variable-length representations, but this discussion is deferred to Chapter **??**, where we will deal with the efficient encoding of integers.

Figure 1.1 provides a running example in which the dictionary strings are stored in an array S, according to an arbitrary order.

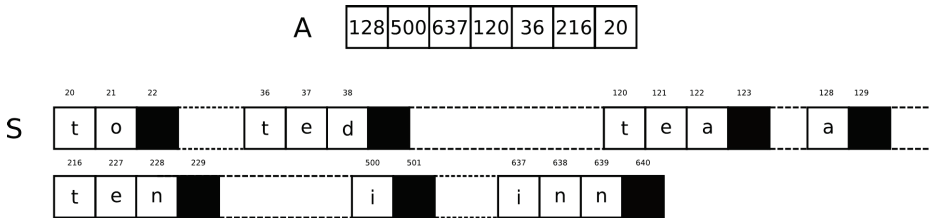

FIGURE 1.1: The array S of strings and the array A of (indirectly) sorted pointers to S's strings.

There are two crucial properties that the sorted array A satisfies:

- all dictionary strings prefixed by P occur contiguously if lexicographically sorted. So their pointers occupy a subarray, say $A[l, r]$, which may be possibly empty if P does not prefix any dictionary string.
- the string P lexicographically precedes $A[l]$ and follows $A[l-1]$.

Since the prefix search returns either the number of dictionary strings prefixed by P, hence the value $r - l + 1$, or visualizes these strings, the key problem is to identify the two extremes l and r efficiently. To this aim, we reduce the prefix search problem to the *lexicographic search* of a pattern Q in $\mathcal{D}$: namely, the search of the lexicographic position of Q among $\mathcal{D}$'s strings. The formation of Q is simple: Q is either the pattern P or the pattern $P\#$, where # is larger than any other alphabet character. It is not difficult to convince yourself that $Q = P$ will precede the string $A[l]$ (see above), whereas $Q = P\#$ will follow the string $A[r]$. This means actually that two lexicographic searches for patterns of length no more than $p + 1$ are enough to solve the prefix-search problem.

The lexicographic search can be implemented by means of an (indirect) binary search over the array A. It consists of $O(\log n)$ steps, each one requiring a string comparison between $P[1, p]$ and the string pointed by the entry tested in A. The comparison is lexicographic and thus takes $O(p)$ time and $O(p/B)$ I/Os, because it may require in the worst case the scan of all $\Theta(p)$ characters of Q.

THEOREM 1.1 *The complexity of a prefix search over the array of string pointers is $O(p \log n)$ time and $O(\frac{p}{B} \log n)$ I/Os, the total space is $L + (1 + w)n$ bytes.*

Proof Time and I/O complexities derive from the previous observations. For the space occupancy, A needs n pointers, each taking a memory word w, and all dictionary strings occupy L bytes plus one-byte delimiter for each of them (commonly ^
0 in C).

The poor time and I/O-complexities derive from the *indirection*, which forces no locality in the memory/string accesses of the binary search. The inefficiency is even more evident if we wish to retrieve all strings prefixed by P, and not just count them. After that the range $A[l, r]$ has been

identified, each string visualization elicits at least one I/O because contiguity in A does not imply contiguity of the pointed strings in S. This may be a major bottleneck if the number of returned strings is large, as it typically occurs in queries that use the prefix search as a preliminary step to select a *candidate set of answers* that have then to be refined via a proper post-filtering process. An example is the solution to the problem of *searching with wild-cards* which involves the presence in P of many special symbols *. The wild-card * matches any substring. In this case if $P = \alpha * \beta * \cdots$, where $\alpha, \beta, \ldots$ are un-empty strings, then we could perform a prefix-search for α in $\mathcal{D}$ and then check brute-forcedly whether P matches the returned strings. This is not an optimal approach, nevertheless it is an easy application of prefix search which puts in evidence how much slow may be in a disk environment a wild-card query if based on the previous approach.

1.1.1 Contiguous allocation of strings

A simple trick to circumvent some of the previous limitations is to store the dictionary strings sorted lexicographically and contiguously on disk. This way (pointers) contiguity in A reflects into (string) contiguity in S. This has two main advantages:

- when the binary search is confined to few strings, they will be closely stored both in A and S, so probably they have been buffered by the system in internal memory (*speed*);
- some compression can be applied to contiguous strings in S, because they typically share some prefix (*space*).

Given that S is stored on disk, we can deploy the first observation by blocking strings into groups of B characters each and then *store* a pointer to the first string of each group in A. The sampled strings are denoted by $\mathcal{D}_B \subseteq \mathcal{D}$, and their number n_B is upper bounded by $\frac{L}{B}$. It is evident that each string in $\mathcal{D}_B$ identifies a block, namely the one it belongs to. Since A has been squeezed to index at most $n_B < n$ strings, the search over A must be changed in order to reflect the *two-level structure* given by the array A and the string-blocks in S. So the idea is to decompose the lexicographic search for Q in a two-stage process: in the first stage, we search for the lexicographic position of Q within the sampled strings of $\mathcal{D}_B$; in the second stage, this position is deployed to identify the block of strings where the lexicographic position of Q lies in. We recall that, in order to implement the prefix search, we have to repeat the above process for the two strings P and $P\#$, so we have proved the following:

THEOREM 1.2 *Prefix search over $\mathcal{D}$ takes $O(\frac{p}{B} \log \frac{L}{B} + \frac{L_{occ}}{B})$ I/OS, where L_{occ} is the length of all the strings prefixed by P in $\mathcal{D}$.*

Proof Once the block of strings $A[i, j]$ prefixed by P has been identified, we can report all of them in $O(\frac{L_{occ}}{B})$ I/Os; scanning the contiguous portion of S that contains those strings.

Since $\frac{L}{B} \leq n$, this solution is faster than the previous one, in addition it can be effectively combined with *Front-Coding compression* to further lowering the space and I/O-complexities.

1.1.2 Front Coding

Given a sequence of sorted strings is probable that adjacent strings share a common prefix. If ℓ is the number of shared characters, then we can substitute them with an encoding of ℓ using a variable-length encoder (see Chapter **??**). This means to replace the initial $\Theta(\ell \log_2 \sigma)$ bits with $O(\log \ell)$ bits, so it is always advantageous in terms of space savings. However its impact depends on the amount of shared characters which, in the case of a dictionary of URLs, can be up to 70%.

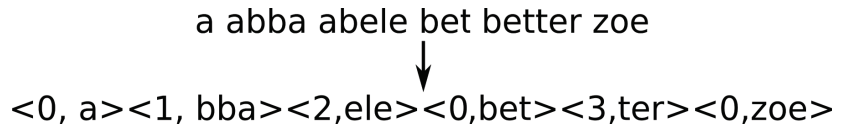

FIGURE 1.2: Front coding (bottom) of few lexicographically-sorted English words (top).

Front coding is a *delta*-compression algorithm, which can be easily defined in an incremental way: given a sequence of strings $(s_1, \ldots, s_n)$, it encodes the string s_i using the couple $(\ell_i, \hat{s}_i)$, where ℓ is the length of the longest common prefix between s_i and its predecessor s_{i-1} (0 if $i = 1$) and $\hat{s}_i = s_i[\ell + 1, |s_i|]$ is the "remaining suffix" of the string s_i (see Figure 1.2). Decoding is symmetric in that, given a pair (ℓ, s), we have to copy ℓ characters from the previous string in the sequence and then append the remaining suffix s: This takes $O(|s|)$ optimal time and $O(1 + \frac{|s|}{B})$ I/Os, provided that the preceding string is available. In general, the reconstruction of s_i may require to scan back the input sequence up to the first string s_1, which is available in its entirety. So we may possibly need to scan $(\hat{s}_1, \ell_1), \ldots, (\hat{s}_{i-1}, \ell_{i-1})$ and reconstruct $s_1, \ldots, s_{i-1}$ in order to decode $(\ell_i, \hat{s}_i)$. Therefore, the time cost to decode s_i might be much higher than $O(|s_i|)$.

For this reason, we apply front-coding to each block of strings in our two-level scheme above by restarting the compression at the beginning of every block, namely, by storing each first string *uncompressed*. This has two immediate advantages: (1) these uncompressed strings are the ones participating in the binary-search process and thus they are immediately available and do not need to be decompressed when compared with Q; (2) each block is compressed individually and thus its scanning for searching Q can be combined with its decompression without incurring in any slowdown. We call this storage scheme "Front-coding with bucketing", and shortly denote it by FC_B. Figure 1.3 provides a running example in which the strings "alcatraz", "alcyone", "ananas", and "astral" are stored explicitly because they are the first of each block.

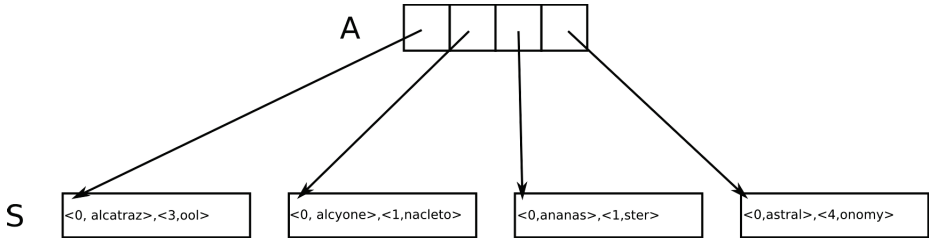

FIGURE 1.3: Two-level indexing of the set of strings $\mathcal{D}$ = `alcatraz, alcool, alcyone, anacleto, ananas, aster, astral, astronomy` are compressed with FC_B, where we assumed that each page is able to store two strings.

As a positive side-effect, this approach reduces the number of sampled strings because it can potentially increase the number of strings stuffed in a disk page: we start from s_1 and we front-compress the strings of $\mathcal{D}$ in order; whenever the compression of a string s_i overflows the current block, it starts a new block where it is stored *uncompressed*. The number of sampled strings lowers from $\left\lceil \frac{L}{B} \right\rceil$ to about $\left\lceil \frac{FC_B(\mathcal{D})}{B} \right\rceil$, where $FC_B(\mathcal{D})$ is the space required by FC_B to store all the dictionary strings. This also impacts onto the number of I/Os needed for a prefix search in a obvious manner:

THEOREM 1.3 *Prefix search over $\mathcal{D}$ takes $O(\frac{p}{B}\log\frac{FC_B(\mathcal{D})}{B} + \frac{FC_B(\mathcal{D}_{occ})}{B})$ I/Os, where $\mathcal{D}_{occ} \subseteq \mathcal{D}$ is the set of strings in the answer set.*

So, in general, compressing the strings is a good idea because it lowers the space required for storing the strings, and the number of I/Os. However it may increase the time complexity of the scan of a block from $O(B)$ to $O(B^2)$ because of the decompression of that block. For example take the sequence of strings $(a, aa, aaa, \ldots)$ which is front coded as $(0, a), (1, a), (2, a), (3, a), \ldots$. In a block we can stuff $\Theta(B)$ pairs (strings) which correspond to strings whose total length is $\sum_{i=0}^{B}\Theta(i) = \Theta(B^2)$ characters. In practice the space reduction consists of a constant factor so the time increase incurred by a block scan is negligible.

Overall this approach introduces a time/space trade-off driven by the block size B: the longer is B, the better is the compression ratio but the slower is a prefix search because of a longer scan-phase; conversely, the shorter is B, the faster is the scan-phase but the worse is the compression ratio because of a larger number of fully-copied strings. We wish to decouple the search from the compression issues making them one independent from the other.

We notice that the proposed data structure consisted of *two* levels: the "upper" level contains references to the *sampled strings* $\mathcal{D}_B$, whereas the "lower" level contains the strings themselves stored in a block-wise fashion. The choice of the algorithms and data structures used in the two levels are "orthogonal" to each other, and thus can be decided independently. It goes without saying that this 2-level scheme for searching-and-storing a dictionary of strings is suitable to be used in a hierarchy of two memory levels, such as the cache and the internal memory. This is typical in Web search, where $\mathcal{D}$ is the dictionary of terms to be searched by the users and disk-accesses have to be avoided in order to support each search over $\mathcal{D}$ in few millisecs.

In the next three sections we propose three improvements to the 2-level solution above, two of them regard the first level of the sampled strings, one concerns with the string storage. Actually, these proposals have an interest in themselves and thus the reader should not confine their use to the one described in these notes.

1.2 Interpolation search

Until now, we have used binary search over the array A of string pointers. But if $\mathcal{D}_B$ satisfies some statistical properties, there are searching schemes which support faster searches, such as the well known *interpolation search*. In what follows we describe a variant of classic interpolation search which offers some interesting additional properties (details at [2]). For simplicity of presentation we describe the algorithm in terms of a dictionary of integers, knowing that it can be generalized to items drawn from an ordered general universe. In the prefix-search problem items are strings, nevertheless we can still look at them as integers in base σ.

So without loss of generality, assume that $\mathcal{D}_B$ is a sequence of integers $x_1 \ldots x_m$ with $x_i < x_{i+1}$ and $m = n_B$, from above. We evenly subdivide the range $[x_1, x_m]$ into m bins $B_1, \ldots, B_m$, each of them representing a range of length $b = \frac{x_m - x_1 + 1}{m}$. Specifically $B_i = [x_1 + (i-1)b, x_1 + ib)$. Figure 1.4 reports an example where $m = 12$, $x_1 = 1$ and $x_{12} = 36$ and thus the bin length is $b = 3$.

The algorithm searches for an integer y in two steps. In the first step it calculates i, the index of the candidate bin B_i where y could occur: $i = \lfloor\frac{y - x_1}{b}\rfloor + 1$. In the second step, it does a binary search in B_i for y, thus taking $O(\log|B_i|) = O(\log b)$ time. The value of b depends on the magnitude of the integers present in the indexed dictionary. Surprisingly enough, we can instead get a cleaner bound as follows:

THEOREM 1.4 *We can search for an integer in a dictionary of size m taking $O(\log\Delta)$ time in the worst case.*

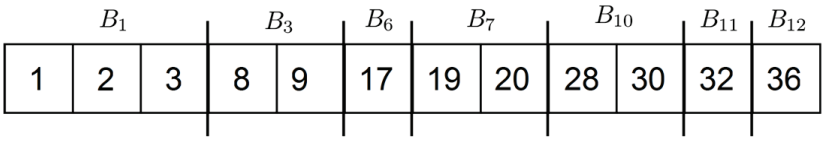

FIGURE 1.4: An example of use of interpolation search over an itemset of size 12. The bins are separated by bars; some bins, such as B_4 and B_8, are empty.

Proof Correctness is immediate. For the time complexity, let us set Δ as the ratio between the maximum and the minimum gap between two consecutive integers:

$$\Delta = \frac{\max_{i=1\ldots m}(x_i - x_{i-1})}{\min_{i=1\ldots m}(x_i - x_{i-1})}$$

Since the maximum of a series of integers is at least large as the mean of those integers, we have that:

$$\max_{i=1\ldots m}(x_i - x_{i-1}) \geq \frac{\sum_{i=1}^{m} x_i - x_{i-1}}{m-1} > \frac{x_m - x_1}{m} \tag{1.1}$$

If the integers are spaced apart by s units, we can pack in a bin at most b/s integers. In our case $s = \min_{i=1,\ldots,m}(x_i - x_{i-1})$, so we can conclude that every bin B_i contains no more than b/s integers. Given the upper bound on the maximum gap, obtained before, we can write:

$$|B_i| \leq b/s < \frac{\max_{i=1\ldots m}(x_i - x_{i-1})}{\min_{i=1,\ldots,m}(x_i - x_{i-1})} = \Delta$$

.

So the theorem follows due to the binary search performed within B_i. ∎

We note the following interesting properties of the proposed algorithm:

- The algorithm is oblivious to the value of Δ, although its complexity can be written in terms of this value.
- The worst-case search time is $O(\log m)$ and thus the precise bound should be $O(\log \min\{\Delta, m\})$. So it cannot be worst than the binary search.
- The algorithm reproduces the $O(\log \log m)$ time performance of classic interpolation search on data drawn independently from the uniform distribution, as shown in the following lemma:

LEMMA 1.1 If the m integers are drawn uniformly, the proposed algorithm takes $O(\lg \lg m)$ time with high probability.

Proof Say integers are uniformly distributed over $[0, U]$. Assume to partition the integers in $r = \frac{m}{2 \log m}$ ranges. We have the probability $1/r$ that an integer belongs to a given range. The probability that a given range does not contain any integer is $(1 - \frac{1}{r})^m = (1 - \frac{2 \log m}{m})^m = O(e^{-2 \log m}) = O(1/m^2)$. So the probability that at least one range remains empty is smaller than $O(1/m)$.

If every range contains at least one integer, with high probability, the maximum distance between two adjacent integers cannot be larger than twice the range's length: namely $\max_i(x_i - x_{i-1}) \leq 2U/r = O(\frac{U \log m}{m})$.

Let us now take $r = \Theta(m \log m)$ ranges, similarly as above we can prove that every adjacent pair of ranges contains at most one integer with high probability. Thus we can lower bound the minimum gap with the length of one range: $\min_i(x_i - x_{i-1}) \geq U/r = \Theta(\frac{U}{m \log m})$. Taking the ratio between the minimum and the maximum gap, we get the desired $\Delta = O(\log^2 m)$. ∎

If this algorithm is applied to our string context, and strings are uniformly distributed, the number of I/Os required to prefix-search P in the dictionary $\mathcal{D}$ is $O(\frac{p}{B} \log\log \frac{L}{B})$. This is an exponential reduction in the search time performance according to the dictionary length.

1.3 Locality-preserving front coding

This is an elegant variant of front coding which provides a controlled trade-off between space occupancy and time to decode one string [1]. The key idea is simple, and thus easily implementable, but proving its guaranteed bounds is tricky. We can state the underlying algorithmic idea as follows: *a string is front-coded only if its decoding time is proportional to its length, otherwise it is written uncompressed*. The outcome in time complexity is clear: we compress only if decoding is optimal. But what appears surprising is that, even if we concentrated on the time-optimality of decoding, its "constant of proportionality" controls also the space occupancy of the compressed strings. It seems magic, indeed it is!

FIGURE 1.5: The two cases occurring in LPFC. Red rectangles are copied strings, green rectangles are front-coded strings.

Formally, suppose that we have front-coded the first $i-1$ strings $(s_1, \ldots, s_{i-1})$ into the compressed sequence $\mathcal{F} = (0, \hat{s}_1), (\ell_2, \hat{s}_2), \ldots, (\ell_{i-1}, \hat{s}_{i-1})$. We want to compress s_i so we scan backward at most $c|s_i|$ characters of $\mathcal{F}$ to check whether these characters are enough to reconstruct s_i. This actually means that an uncompressed string is included in those characters, because we have available the first character for s_i. If so, we front-compress s_i into $(\ell_i, \hat{s}_i)$; otherwise s_i is copied uncompressed in $\mathcal{F}$. The key difficulty here is to show that the strings which are left uncompressed, and were instead compressed by the classic front-coding scheme, have a length that can be controlled by means of the parameter c as the following theorem shows:

THEOREM 1.5 *Locality-preserving front coding takes at most* $(1+\epsilon)FC(\mathcal{D})$ *space, and supports the decoding of any dictionary string* s_i *in* $O(\frac{|s_i|}{\epsilon B})$ *optimal I/Os.*

Proof We call any uncompressed string s, a *copied* string, and denote the $c|s|$ characters explored during the backward check as the *left extent* of s. Notice that if s is a copied string, there can be no copied string preceding s and beginning in its left extent (otherwise it would have been front-

coded). Moreover, the copied string that precedes S may *end* within s's left extent. For the sake of presentation we call *FC-characters* the ones belonging to the output suffix of a front-coded string.

Clearly the space occupied by the front-coded strings is upper bounded by $FC(\mathcal{D})$. We wish to show that the space occupied by the copied strings, which were possibly compressed by the classic front-coding but are left uncompressed here, sums up to $\epsilon\ FC(\mathcal{D})$, where ϵ is a parameter depending on c.

We consider two cases for the copied strings depending on the amount of FC-characters that lie between two consecutive occurrences of them. The first case is called *uncrowded* and occurs when that number of FC-characters is at least $\frac{c|s|}{2}$; the second case is called *crowded*, and occurs when the number of FC-characters is at most $\frac{c|s|}{2}$. Figure 1.6 provides an example which clearly shows that if the copied string s is crowded then $|s'| \geq c|s|/2$. In fact, s' starts before the left extent of s but ends within the last $c|s|/2$ characters of that extent. Since the extent is $c|s|$ characters long, the above observation follows. If s is uncrowded, then it is preceded by at least $c|s|/2$ characters of front-coded strings (FC-characters).

FIGURE 1.6: The two cases occurring in LPFC. The green rectangles denote the front-coded strings, and thus their FC-characters, the red rectangles denote the two consecutive copied strings.

We are now ready to bound the total length of copied strings. We partition them into chains composed by one uncrowded copied-string followed by the maximal sequence of crowded copied-strings. In what follows we prove that the total number of characters in each chain is proportional to the length of its first copied-string, namely the uncrowded one. Precisely, consider the chain $w_1 w_2 \cdots w_x$ of consecutive copied strings, where w_1 is uncrowded and the following w_is are crowded. Take any crowded w_i. By the observation above, we have that $|w_{i-1}| \geq c|w_i|/2$ or, equivalently, $|w_i| \leq 2|w_{i-1}|/c = \cdots = (2/c)^{i-1}|w_1|$. So if $c > 2$ the crowded copied strings shrink by a constant factor. We have $\sum_i |w_i| = |w_1| + \sum_{i>1} |w_i| = |w_1| + \sum_{i>1}(2/c)^{i-1}|w_1| = |w_1| \sum_{i\geq 0}(2/c)^i < \frac{c|w_1|}{c-2}$.

Finally, since w_1 is uncrowded, it is preceded by at least $c|w_1|/2$ FC-characters (see above). Since the total number of these FC-characters is bounded by $FC(\mathcal{D})$, we can upper bound the total length of the uncrowded strings by $(2/c)FC(\mathcal{D})$. By plugging this into the previous bound on the total length of the chains, we get $\frac{c}{c-2} \times \frac{2FC(\mathcal{D})}{c} = \frac{2}{c-2} FC(\mathcal{D})$. The theorem follows by setting $\epsilon = \frac{2}{c-2}$. ∎

So locality-preserving front coding (shortly LPFC) is a compressed storage scheme for strings that can substitute their plain storage without introducing any asymptotic slowdown in the accesses to the compressed strings. In this sense it can be considered as a sort of *space booster* for any string indexing technique.

The two-level indexing data-structure described in the previous sections can benefit of LPFC as follows. We can use A to point to the uncompressed strings of LPFC. This way the buckets delimited by the sampled strings have variable length, but this length is proportional to the length of the included strings. Scanning then takes time proportional to the returned string, and hence is optimal. So if A fits within the internal-memory space allocated by the programmer, or available in

cache, then this approach is efficient. Otherwise, we could deploy our two-level blocking scheme to index the uncompressed strings of LPFC. The advantage would be that we are reducing the overall number of strings to which it is applied, hence potentially limiting its weaknesses in time/IO efficiency and space occupancy.

In the next sections we propose a trie-based approach that takes full-advantage of LPFC resulting efficient in time, I/Os and space.

1.4 Compacted Trie

We already talked about tries in Chapter **??**, here we dig further into their properties as efficient search data structures. In our context, the trie is used for the indexing of the sampled strings $\mathcal{D}_B$ in internal memory. This induces a speed up in the first stage of the prefix search from $O(\log(L/B))$ to $O(p)$ time, thus resulting surprisingly independent of the dictionary size. The reason is the power of the RAM model which allows to manage and address memory-cells of $O(\log n)$ bits in constant time.

A trie is a multi-way tree whose edges are labeled by characters of the indexed strings. An internal node u is associated with a string $s[u]$ which is indeed a *prefix* of a dictionary string. String $s[u]$ is obtained by concatenating the characters found on the downward path that connects the trie's root with the node u. A leaf is associated with a dictionary string. All leaves which descend from a node u are prefixed by $s[u]$. The trie has n leaves and L nodes, one per string character. Figure **??** provides an illustrative example of a trie built over 6 strings. The substrings showed in the nodes are illustrated just for clarity, they do not need to be stored because one can reconstruct them during the downward traversal of the tree structure. This form of trie is commonly called *uncompacted* because it can have *unary paths*, such as the one leading to string `inn`.[1]

If we want to check if a string P prefixes some dictionary string, we have just to check if there is a downward path spelling out P. All leaves descending from the reached node provide the correct answer to our prefix search. So tries do not need the reduction to the lexicographic search operation, introduced for the binary-search approach.

A big issue is how to efficiently find the "edge to follow" during the downward traversal of the trie, because this impacts onto the overall efficiency of the pattern search. The efficiency of this step hinges on a proper storage of the edges (and their labeling characters) outgoing from a node. The simplest data structure that does the job is the *linked list*. Its space requirement is optimal, namely proportional to the number of outgoing edges, but it incurs in a $O(\sigma)$ cost per traversed node. The result would be a prefix search taking $O(p\ \sigma)$ time in the worst case, which is too much for large alphabets. If we store the branching characters (and their edges) into a sorted array, then we could binary search it, taking $O(\log \sigma)$ time per node. A faster approach could be to use a full-sized array of σ entries, the un-empty ones are the entries corresponding to the existing branching characters. The time to branch out of a node would be $O(1)$ and thus $O(p)$ for the overall prefix search, but the space occupancy of the trie would grow up to $O(\sigma L)$, which may be unacceptably high for large alphabets.

The best approach consists of resorting a *perfect hash table*, which would guarantee constant access time and optimal space occupancy, thus combining the best of the two previous solutions. For details about perfect hashes we refer the reader to Chapter **??**.

[1] The trie cannot index strings which are one the prefix of the other. The former string would end up into an internal node. To avoid this case, each string is extended with a special character which is not present in the alphabet, say \$.

THEOREM 1.6 *The trie solves the prefix-search problem in $O(p + L_{occ})$ time. The trie consists of L nodes, and thus takes $O(L)$ space.*

Proof Let u be the node such that $s[u] = P$. All strings descending from u are prefixed by P. The visit of the subtree descending from u would visualize the strings prefixed by P. ∎

A Trie can be wasteful in space if there are long strings with a short common prefix: this would induce a significant number of unary nodes. We can save space by *contracting* the unary paths into one single edge. This way edge labels become (possibly long) sub-strings rather than characters, and the resulting trie is named *compacted*. Figure 1.7 (left) shows an example of compacted trie. It is evident that each edge-label is a substring of a dictionary string, say $s[i, j]$, so it can be represented via a triple $\langle s, i, j\rangle$. Given that each node is at least binary, the number of internal nodes and edges is $O(n)$. So the total space required by a compacted trie is $O(n)$ too.

Prefix searching is implemented similarly as done for uncompacted tries. The difference is that it alternates character-branches out of internal nodes, and sub-string matches with edge labels. If the edges spurring from the internal nodes are again implemented with perfect hast tables, we get:

THEOREM 1.7 *The trie solves the prefix-search problem in $O(p + L_{occ})$ time. The trie consists of $O(n)$ nodes, and thus its storage takes $O(n)$ space. It goes without saying that the trie needs also the storage of the dictionary strings to resolve its edge labels, hence additional L space.*

At this point an attentive reader can realize that the compacted trie can be used also to search for the lexicographic position of a string Q among the indexed strings. It is enough to percolate a downward path spelling Q as much as possible until a mismatch character is encountered. This character can then be deployed to determine the lexicographic position of Q, depending on whether the percolation stopped in the middle of an edge or in a trie node. So the compacted trie is an interesting substitute for the array A in our two-level indexing structure and could be used to support the search for the candidate bucket where the string Q occurs in, taking $O(p)$ time in the worst case. Since each traversed edge can induce one I/O, to fetch its labeling substring to be compared with the corresponding one in Q, we point out that this approach is efficient if the trie and its indexed strings can be fit in internal memory.

Otherwise it presents two main problems: the linear dependance of the I/Os on the pattern length p, and the space dependance on the block-size B. The *Patricia Trie* solves the former problem, and its combination with the LPFC provides an efficient solution to both problems.

1.5 Patricia Trie

A Patricia Trie built on a string dictionary is a compacted Trie in which the edge labels consist just of their initial *single characters*, and the internal nodes are labeled with integers denoting the *lengths* of the associated strings. Figure 1.7 illustrates how to convert a Compacted Trie (left) into a Patricia Trie (right).

Even if the Patricia Trie strips out some information from the Compacted Trie, it is still able to support the search for the lexicographic position of a pattern P among a (sorted) sequence of strings, with the significant advantage (discussed below) that this search needs to access only one single string, and hence execute typically one I/O instead of the p I/Os potentially incurred by the edge-resolution in compacted tries. This algorithm is called *blind search* in the literature [3]. It is a little bit more complicated than the one in classic tries, because of the presence of only one character per edge label, and in fact consists of two stages:

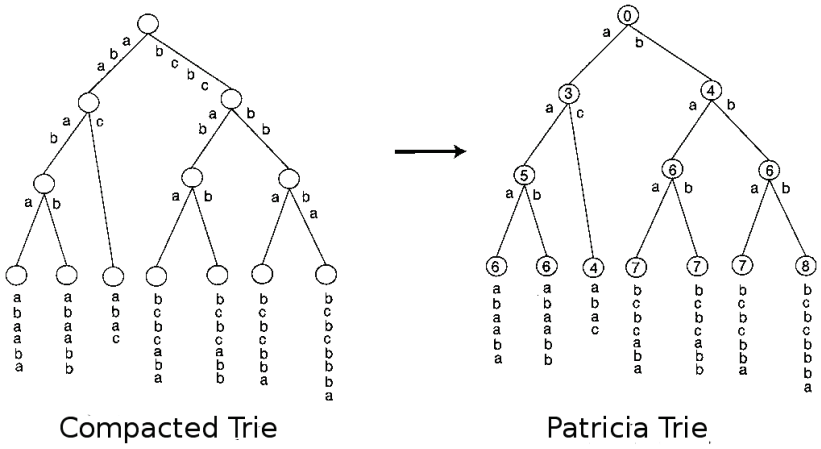

FIGURE 1.7: An example of Compacted Trie and the corresponding Patricia Trie.

- Trace a downward path in the Patricia Trie to locate a leaf l which points to an interesting string of the indexed dictionary. This string does not necessarily identify P's lexicographic position in the dictionary (which is our goal), but it provides *enough information* to find that position in the second stage. The retrieval of the interesting leaf l is done by traversing the Patricia Trie from the root and comparing the characters of P with the single characters which label the traversed edges until either a leaf is reached or no further branching is possible. In this last case, we choose l to be any descendant leaf from the last traversed node.
- Compare P against the string pointed by leaf l, in order to determine their longest common prefix. Let ℓ be its length, then it is possible to prove that (see [3]) the leaf l stores one of the strings indexed by the Patricia Trie that shares the longest common prefix with P. The length ℓ and the mismatch character $P[\ell + 1]$ are then used in two ways: first to determine the edge where the mismatch character lies; and then, to select the leaf descending from that edge which identifies the lexicographic position of P within the (sorted) dictionary and, thus, among the leaves of the Patricia Trie.

A running example is illustrated in Figure 1.8.

In order to understand why the algorithm is correct, let us take the path spelling out the string $P[1, \ell]$. We have two cases, either we reached an internal node u such that $|s[u]| = \ell$ or we are in the middle of an edge (u, v), where $|s[u]| < \ell < |s[v]|$. In the former case, all strings descending from u are the ones in the dictionary which share ℓ characters with the pattern, and this is the *lcp*. The correct lexicographic position therefore falls among them or is adjacent to them, and thus it can be found by looking at the branching characters of the edges outgoing from the node u. This is correctly done also by the blind search that surely stops at u, computes ℓ and finally determines the correct position of P by comparing u's branching characters against $P[\ell + 1]$.

In the latter case the blind search reaches v by skipping the mismatch character on (u, v), and possibly goes further down in the trie because of the possible match between branching characters and further characters of P. Eventually a leaf descending from v is taken, and thus ℓ is computed correctly given that all leaves descending from v share ℓ characters with P. So the backward traversal executed in the second stage of the Blind search reaches correctly the edge (u, v), which is above the selected leaf. There we deploy the mismatch character which allows to choose the correct

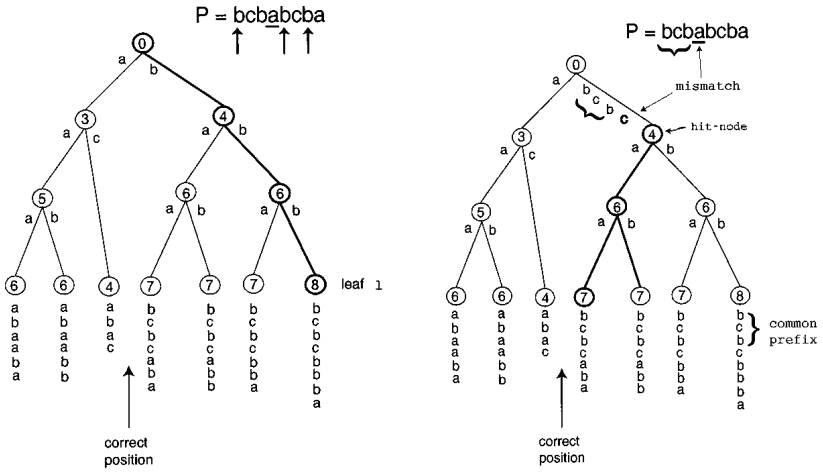

FIGURE 1.8: An example of the first (left) and second (right) stages of the blind search for P in a dictionary of 7 strings.

lexicographic position of P which is either to the left of the leaves descending from v or to their right. Indeed all those leaves share $|s[v]| > \ell$ characters, and thus P falls adjacent to them, either to their left or to their right. The choice depends on the comparison between the two characters $P[\ell + 1]$ and $s[v][\ell + 1]$.
The blind search has excellent performance:

THEOREM 1.8 *A Patricia trie takes $O(n)$ space, hence $O(1)$ space per indexed string, and the blind search for a pattern $P[1, p]$ requires $O(p)$ time to traverse the trie's structure and compares one single string (possibly on disk).*

This theorem states that if $n < M$ then we can stuff the Patricia trie in the internal memory of our PC, store the dictionary strings on disk, and then supports the prefix search for a pattern P in $O(p)$ time and $O(p/B)$ I/Os. The total required space would be the one needed to stored the strings, and thus $O(L)$. If we wish to compress the dictionary strings, then we need to resort front-coding.

More precisely, we combine the Patricia Trie and LPFC as follows. We fit in the memory of our commodity PC the Patricia trie of the dictionary $\mathcal{D}$, and store on disk the locality-preserving front coding of the dictionary strings. The traversals of the Patricia trie take $O(p)$ time and no I/Os (Theorem 1.8), because can be executed entirely in internal memory. Conversely the computation of the lcp takes $O(|s|/B)$ I/Os, because it needs to decode a string s sharing the lcp with P from its LPFC-representation (Theorem 1.5). Once the lexicographic position of P is determined among the leaves of the Patricia trie, we have to decode the strings in $\mathcal{D}_{occ}$, which takes optimal time and I/Os because of the properties of their LPFC-encoding.

THEOREM 1.9 *The data structure composed of the Patricia Trie as the index in internal memory ("upper level") and the LPFC for storing the strings on disk ("lower level") requires $O(n)$ space in memory and $O((1 + \epsilon)FC(\mathcal{D}))$ on disk. Furthermore, a prefix search of P requires $O(\frac{p}{B} + \frac{|s|}{B\epsilon} + \frac{(1+\epsilon)FC(\mathcal{D}_{occ})}{B})$ I/Os, where:*

- *s is the "interesting string" determined in the first stage of the Blind search;*

- $\mathcal{D}_{occ} \subseteq \mathcal{D}$ *is the set of strings prefixed by* P.

1.6 String B-Tree$^{\infty}$

The final question we address in this lecture is: What if $n = \Omega(M)$, so that the Patricia trie is too big to be fit in the internal memory of our PC ? In this case we cannot store on disk the Patricia Trie because it would take $\Omega(p)$ I/Os in the two traversals performed by the Blind search. So we need to find a different approach, which was devised in [3] by proposing the so called *String B-Tree* data structure.

The key idea consists of dividing the big Patricia trie into a set of smaller Patricia tries, each fitting into one disk page. And then linking together all of them in a B-Tree structure. Below we outline a constructive definition of the String B-Tree, for details on this structure and the supported operations we refer the interested reader to the cited literature.

We partition the dictionary $\mathcal{D}$ into a set of smaller, equally sized chunks $\mathcal{D}_1, \ldots, \mathcal{D}_m$, each including $\Theta(B)$ strings. This way, we can index each chunk $\mathcal{D}_i$ with a Patricia Trie that fits into one disk page and embed it into a leaf of the B-Tree. In order to search for P among those set of nodes, we take from each partition $\mathcal{D}_i$ its *first* and *last* (lexicographically speaking) strings s_{if} and s_{il}, defining the set $\mathcal{D}^1 = \left\{s_{1f}, s_{1l}, \ldots, s_{mf}, s_{ml}\right\}$.

Recall that the prefix search for P boils down to the lexicographic search of a pattern Q, properly defined from P. If we search Q within $\mathcal{D}^1$, we can discover one of the following three cases:

1. Q falls before the first or after the last string of $\mathcal{D}$, if $Q < s_{1f}$ or $Q > s_{ml}$.
2. Q falls among the strings of some $\mathcal{D}_i$, and indeed it is $s_{if} < Q < s_{il}$. So the search is continued in the Patricia trie that indexes $\mathcal{D}_i$;
3. Q falls between two chunks, say $\mathcal{D}_i$ and $\mathcal{D}_{i+1}$, and indeed it is $s_{il} < Q < s_{(i+1)f}$. So we found Q's lexicographic position between these two chunks.

In order to establish which of the three cases occurs, we need to search efficiently in $\mathcal{D}^1$ for the lexicographic position of Q. Now, if $\mathcal{D}^1$ is small and can be fit in memory, we can build on it a Patricia trie ad we are done. Otherwise we repeat the partition process on $\mathcal{D}^1$ to build a smaller set $\mathcal{D}^2$, in which we sample, as before, two strings every B, so that $|\mathcal{D}^2| = \frac{2|\mathcal{D}^1|}{B}$. We continue this partitioning process for k steps, until it is $|\mathcal{D}^k| = O(B)$ and thus we can fit the Patricia trie built on $\mathcal{D}^k$ within one disk page[2].

We notice that each disk page gets an even number of strings when partitioning $\mathcal{D}^1, \ldots, \mathcal{D}^k$, and to each pair $\left(s_{if}, s_{il}\right)$ we associate a pointer to the block of strings which they delimit. The final result of the process is then a B-Tree over string pointers. The *arity* of the tree is $\Theta(B)$, because we index $\Theta(B)$ strings in a single node. The nodes of the String B-Tree are then stored on disk. The following Figure 1.9 provides an illustrative example for a string B-tree built over 7 strings.

A (prefix) search for the string P in a String B-Tree is simply the traversal of the B-Tree, which executes at each node a lexicographic search of the proper pattern Q in the Patricia trie of that node. This search discovers one of the three cases mentioned above, in particular:

- case 1 can only happen on the root node;
- case 2 implies that we have to follow the node pointer associated to the identified partition.

[2] Actually, we could stop as soon as $\left|\mathcal{D}^k\right| = O(M)$, but we prefer the former to get a standard B-Tree structure.

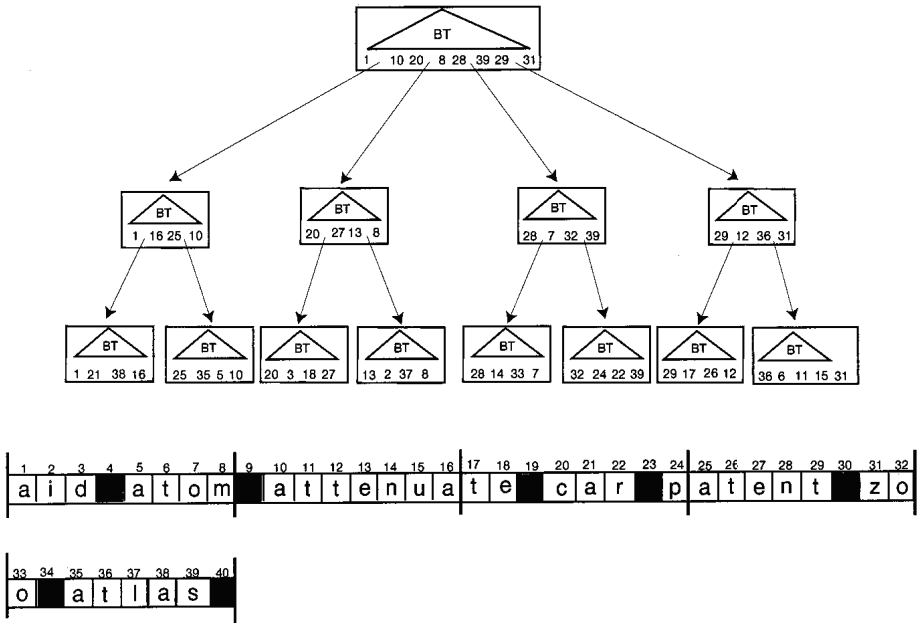

FIGURE 1.9:
An example of an String B-tree on built on the suffixes of the strings in $\mathcal{D}$ = {*'aid'*, *'atlas'*, *'atom'*, *'attenuate'*, *'car'*, *'patent'*, *'zoo'*}.

The I/O complexity of the data structure just defined is pretty good: since the arity of the B-Tree is $\Theta(B)$, we have $\Theta(\log_B n)$ levels, so a search traverses $\Theta(\log_B n)$ nodes. Since on each node we need to load the node's page into memory and perform a Blind search over its Patricia trie, we pay $O(1 + \frac{p}{B})$ I/Os, and thus $O(\frac{p}{B} \log_B n)$ I/Os for the overall prefix search of P in the dictionary $\mathcal{D}$.

This result is good but not yet *optimal*. The issue that we have to resolve in to reach optimality is *rescanning*: each time we do a Blind search, we compare Q and one of the strings stored in the currently visited B-Tree node from their beginning. However, as we go down in the string B-tree we can capitalize on the scanned characters of Q and thus avoid the rescan of these characters during the subsequent lcp-computations. So if f characters have been already matched in Q during some previous lcp-computation, the next lcp-computation can compare Q with a dictionary string by starting from their $(f + 1)$-th character. The cons of this approach is that strings have to be stored uncompressed, in order to support the efficient access to that $(f + 1)$-th character. Working out all the details [3], one can show that:

THEOREM 1.10 *A prefix search in the String B-Tree built over the dictionary $\mathcal{D}$ takes $O(\frac{p+L_{occ}}{B} + \log_B n)$ I/Os, where L_{occ} is the total length of the dictionary strings which are prefixed by P. The data structure occupies $O(\frac{L}{B})$ disk pages and, indeed, strings are stored uncompressed on disk.*

If we want to store the strings compressed on disk, we cannot just plug LPFC in the approach illustrated above, because the decoding of LPFC works only on full strings, and thus it does not support the efficient skip of some characters without wholly decoding the compared string. [1] discusses a sophisticated solution to this problem which gets the I/O-bounds in Theorem 1.10 but

in the cache-oblivious model and guaranteeing LPFC-compressed space. We refer the interested reader to that paper for details.

We conclude this chapter by sketching a solution to the substring search problem, as we left it unsolved in these notes. The substring search of a pattern P can be implemented easily by indexing in a String B-Tree the set $SUF(\mathcal{D})$ of all the suffixes of the dictionary strings and then performing on this set a prefix search of P. The final result is:

THEOREM 1.11 *A substring search in the String B-Tree built upon $SUF(\mathcal{D})$ takes $O(\frac{p+L_{occ}}{B} + \log_B L)$ I/Os, where L_{occ} is the total length of the dictionary strings that have P as a substring.*

Proof The bound derives directly from theorem 1.10 noting that, since there is a bijection between the strings in $SUF(\mathcal{D})$ and the characters of the dictionary strings, $|SUF(\mathcal{D})| = L$. ∎

References

[1] Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. Cache-oblivious string B-trees. In *Procs ACM Symposium on Principles of Database Systems*, pages 223–242, 2006.

[2] Erik D. Demaine, Thouis Jones, and Mihai Pătraşcu. Interpolation search for non-independent data. In *Procs ACM-SIAM Symposium on Discrete algorithms*, pages 529–530, 2004.

[3] Paolo Ferragina and Roberto Grossi. The String B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.