

---

Advanced Database Technology  
Anna Östlin Pagh and Rasmus Pagh  
IT University of Copenhagen  
Spring 2004

## **Text indexing**

April 22, 2004

Based on  
[KärkkäinenRao03]

---

# — Text indexing

---

Data in form of text:

- Digital libraries
- WWW, HTML-pages
- Biological data (e.g. DNA)
- XML
- etc.

We want to:

- Search for specific strings
- Compare strings
- Analyze text data

## — Problem session —

---

Can relational databases be used for text data? Think about it by considering the following two situations:

- We have a number of HTML-pages for which we want a database so that we can search for all pages containing a specific key-word. How can the HTML-pages be stored in a relational database and what query will return the desired answer?
- We have a collection of DNA strings and we want a database that supports queries of the form “Is ATTG...T a substring of any of the DNA strings in the database?”. How can this be implemented using a relational database?

# — This lecture

---

- Problem definition
- Internal memory data structures
  - Suffix trees
  - Suffix array
- External memory data structures
  - Patricia trie and Pat tree
  - Short Pat array
  - String B-tree
- Google's text indexing technique (anno 1998)

# — Terminology

---

An **alphabet**, denoted  $\Sigma$ , is a set of (ordered) characters.

A **string**  $S$  is an array of characters,  $S[1, n] = S[1]S[2] \cdots S[n]$ .

$S[i, j] = S[i] \cdots S[j]$  is a **substring** of  $S$ .

$S[1, j]$  is a **prefix** of  $S$ .  $S[i, n]$  is a **suffix** of  $S$ .

$\Sigma^*$  denotes all strings over alphabet  $\Sigma$ .

We can sort strings according to **lexicographic order**.

Example: For  $\Sigma = \{a, b, c, \dots, z\}$ , where  $a < b < c < \cdots < z$ , the lexicographic order is the same as in a dictionary.

## — Problem definition —

---

### Indexed String Matching Problem

Let  $T$  be a set of  $K$  strings in  $\Sigma^*$ , where  $N$  is the total length of all strings in  $T$ .

**String matching query on  $T$ :** Given a pattern  $P$  find all occurrences of  $P$  in the strings in  $T$ .

**Static problem:** Store  $T$  in a data structure that support string matching queries. Such a data structure is called a **full-text index**.

**Dynamic version:** Supports also insertions and deletions in the full-text index.

## — Suffixes and prefixes —

---

### **An important observation:**

For all occurrences of a pattern  $P$  in a text  $T$  there is a suffix of  $T$  that has a prefix equal to  $P$ .

## Suffix array

---

A suffix array of a text  $T$  is denoted  $SA_T$ .

An array with one entry for each suffix in  $T$ .

The suffixes are sorted. Pointers to the suffixes of  $T$  are stored (in sorted order) in the array.

Note that all suffixes with the same prefix are stored in consecutive order in the array.

Searching in a suffix array:

Binary search.  $O(\log N)$  string comparisons in at most  $O(|P| \log N)$  time.

Possible to do it faster:  $O(|P| + \log N)$  [Manber and Myers].

Construction:

In time  $O(N \log N)$ .

# Trie

---

- A rooted tree with edges labeled by characters.
- A node in the tree represent the string spelled out following the path from the root to the node.
- A trie for a set  $T$  of strings is the minimal trie, such that for all strings  $t \in T$  there is a node in the trie representing  $t$ .

**Compact trie:** Trie where paths without branches are replaced with one edge labeled with the string labeling the path.

# Suffix trees

---

Data structure for strings with many applications.

For example:

Given a string  $S$  preprocess  $S$  such that the query

- Is  $P$  a substring of  $S$ ?

for any string  $P$  can be answered in time proportional to  $|P|$  and not  $|S|$ .

**Suffix trees:**  $O(|S|)$  time preprocessing and  $O(|P|)$  time for queries.

# — Definition of suffix trees —

---

## Definition:

- $S$  string of length  $m$ .
- Rooted tree with  $m$  leaves labeled  $1, \dots, m$ .
- Internal nodes have degree 2 or more (except possibly the root).
- Edges are labeled with nonempty substrings of  $S$ .
- Two edges out of the same node are not labeled by the same first character.
- For leaf labeled  $i$  the concatenation of the edge labels on the root-to-leaf path spells out the suffix starting at position  $i$ , i.e.  $S[i..m]$ .

## — Suffix trees in external memory —

To get linear space for the suffix tree, the strings can not be stored in the tree. Instead pointers to the string are stored. This is a **problem** in external memory.

### Problem session

- How much space would be needed if the strings were stored along with the edges in the suffix tree?
- Why is it a problem to use pointers to the string when we think about external memory?

## — Patricia trie and Pat tree —

---

A **Patricia trie** is a compact trie, except that the edges are labeled in an other way.

- An edge is labeled by only the first character of the label in the compact trie and the length of the string labeling the edge in the compact trie.
- In the leaves there are pointers to the strings.

A **Pat tree** for a text  $T$  is denoted  $PT_T$ . It is the Patricia trie for the set of suffixes in  $T$ .

## — Searching in a Pat tree —

---

### **Idea:**

When searching in the Pat tree you do not have to access the text more than once. Therefore it is (often) much better for external memory.

### **Time to search:**

$O(|P| + Z)$ , where  $Z$  is the size of the answer.

## — Searching in a Pat tree —

---

### How to search:

Search is almost like in the suffix tree.

- When searching for pattern  $P$  compare the first character in  $P$  with the characters labeling the edges to the children of the root. Skip the following  $x - 1$  characters where  $x$  is the number labeling the edge with the same character as  $P$ . Continue recursively.
- When reaching the end of the pattern  $P$  we know that all leaves in that subtree has the same prefix. Look at one of them and compare to  $P$ . If it matches  $P$ , then all matches  $P$ , otherwise none matches  $P$ .

## — Short Pat array (SPat array) —

---

External memory version of a suffix array.

### Idea:

- Divide the suffix array into blocks of equal size, say size  $p$ .
- Take one (the last) string in each block
- Let these strings form a new shorter array. Store only the first  $l$  characters in each string in the array.
- Choose the block size  $p$  and length  $l$  so that this SPat array fits in memory.
- (More levels can be used if the blocks are too large to fit in one block on disk. Assume for now that  $p$  is not larger than the “real” block size.)

## — Searching a SPat array —

---

The SPat array can be searched (using, e.g., binary search) without extra I/O's.

- If the pattern  $P$  matches more than one string in the SPat array in the first  $l$  characters, then we need to look at the actual strings. Let  $r$  be the number of strings in the SPat array matching  $P$ .  $O(\log r)$  I/O's are needed to find the block to search in.
- When the block to continue the search in is known we need to search the block. A binary search among the  $p$  strings requires  $O(\log p)$  I/O's since we need to look up the string each time to compare the strings.

## — Problem session —

---

What are the two main problems with using a B-tree for storing strings of variable length?

# String B-trees

---

String B-trees combines B-trees and Patricia tries to overcome the problems with comparing strings and storing strings in the nodes of a B-tree.

## Definition:

- The nodes in the tree stores a Patricia trie for the set of strings consisting of the lexicographic smallest and largest strings in all its subtrees.
- The degree of a node is between  $b/2$  and  $b$  (except for the root).
- The strings in the leaves are lexicographic sorted from left to right.

## — Searching in String B-trees —

---

The essential step is to use the Patricia trie to find the right subtree to continue the search in.

**Two cases:**

**Cases 1:**

- The pattern  $P$  matches a string  $S_i$  in the Patricia trie.  
We are done. Scan the leaves left and right of  $S_i$  to find all occurrences of  $P$ .

## — Searching in String B-trees (cont.) —

### Cases 2:

- The pattern has a mismatch in the Patricia trie.

**Note 1:** All strings below the mismatch have the same prefix above the mismatch.

**Note 2:** The pattern  $P$  may have a mismatch before the mismatch in the Patricia trie, since only the first character after a branching point is compared to  $P$ .

Compare  $P$  to some string in the subtree below the mismatch. From this information we know between which strings, in the Patricia trie,  $P$  belongs. We know where to continue to search (or that  $P$  doesn't match any string in the tree).

### Time:

$O(\text{scan}(|P| + Z) + \text{search}(N))$ , where  $\text{scan}(X) = \Theta(X/B)$  and  $\text{search}(N) = \Theta(\log_B N)$ .

## — Updates in String B-trees —

### **Insertion:**

Similar to insertion in B-trees, but now we must insert new strings into the Patricia tries in the nodes.

Insertion of a string  $S$  can be done in  $O(\text{scan}(|S|) + \text{search}(N))$  I/O's, where  $N$  is the number of strings in the String B-tree.

The naive way of doing it takes  $O(\text{scan}(|S|))$  I/O's in each node that is changed.

$O(\text{scan}(|S|) + \text{search}(N))$  is also the amortized cost using the naive algorithm, since the amortized number of split nodes is constant.

### **Deletion:**

Similar.

# — Summary of external data structures —

## Patricia trie and Pat tree:

$O(|P| + Z)$  I/O's (naive implementation).

Efficient if the tree fits in main memory, i.e.  $Scan(|P|)$  I/O's.

Used in String B-trees.

## Short Pat array:

Efficient if enough memory to store the SPat array and if the SPat array does not contain too many strings with equal first  $l$  characters.

## String B-trees:

Searching in  $O(scan(|S| + Z) + search(N))$  I/O's.

Updates in  $O(scan(|S|) + search(N))$  I/O's.

Space needed:  $\Theta(N/B)$  blocks.

Can also be used for dynamic indexed string matching.