External Memory Algorithms and Data Structures

Christian Sommer
Overview

- Application
- Definitions, Computational Model
- Internal Memory Techniques
- External Memory Techniques
  - Pat Trees
  - String B-trees
  - Self-adjusting Skip List
Application

String DB

- Patent DB
- online libraries
- biological DB
- XML DB
- product catalogs
- ...

Full text indexing, Christian Sommer, WS 04/05
Definitions

**Alphabet** $\Sigma$

- finite ordered set of characters
- size $|\Sigma|$ (size of alphabet)
- Constant alphabet model: dictionary operations on sets of characters can be performed in constant time and linear space (approximation with techniques like hashing)

**String, Substring, Prefix, Suffix, Text**

- Substring of $S$: $S[i, j] = S[i] \ldots S[j]$ ($1 \leq i \leq j \leq n$)
- Prefix of $S$: $S[1, k]$
- Suffix of $S$: $S[l, n]$
- Text $T$: set of $K$ strings in $\Sigma^*$, total length $N$
## Definitions [contd.]

- **Full-text index**
  - Data structure storing a text $\mathcal{T}$
  - supporting string matching queries
  - Dynamic version: support insertion and deletions of strings $S$ (size $|S|$) into/from $\mathcal{T}$ (Dictionary operations)

- **String matching queries**
  - Given pattern string $P \in \Sigma^*$ (length $|P|$)
  - Find all occurrences of $P$ as a substring of the strings in $\mathcal{T}$

- **String sorting**
  - Sort a set $S$ of $K$ strings in $\Sigma^*$ in lexicographic order $\leq_L$
Computational model

Parameters

- Problem size $N$: total number of characters in the text
- Memory size $M$: number of characters that fit into internal memory
- Block size $B$: number of characters that fit into a disk block
- $K$: number of strings in the text/set to be sorted
- $R$: size of the answer

Notations

- $\text{Scan}(N) = \Theta \left( \frac{N}{B} \right)$
- $\text{Sort}(N) = \Theta \left( \frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B} \right)$
- $\text{Search}(N) = \Theta(\log_B N)$
**Internal Memory Techniques: Suffix array**

- **Observation:**
  
  occurrence of a pattern \( P \) starts at position \( i \) in a string \( S \in \mathcal{T} \Rightarrow P \) is a prefix of the suffix \( S[i, |S|] \)

- **Example**

  Text \( \mathcal{T} = "\text{String representation}" \)
  
  \( S_1 = "\text{String}" , S_2 = "\text{representation}" \)
  
  Pattern \( P = "\text{present}" \)
  
  \( \Rightarrow i = 3 , S_2[3, |S_2|] = "\text{presentation}" \)

- **Suffix array \( SA_{\mathcal{T}} \)**

  - answers a prefix search query in \( \mathcal{O}(|P| \cdot \log_2 K) \)
  - sorted array of pointers to the suffixes of \( \mathcal{T} \), string matching is done with a binary search, \( \mathcal{O}(\log_2 K) \) string comparisons
  - comparing two strings: \( \mathcal{O}(|P|) \)
$\mathcal{T} = \{\text{banana}\}$

$\Rightarrow \quad \text{SA}_\mathcal{T}$

<table>
<thead>
<tr>
<th></th>
<th>$\text{SA}_\mathcal{T}$</th>
<th>$\text{SA}_\mathcal{T}^{-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>$a$</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>$\text{ana}$</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>$\text{anana}$</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>$\text{banana}$</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>$na$</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>$\text{nana}$</td>
<td>1</td>
</tr>
</tbody>
</table>

$\Rightarrow \quad \text{SA}_\mathcal{T}$

$\Rightarrow \quad \text{SA}_\mathcal{T}^{-1}$

<table>
<thead>
<tr>
<th></th>
<th>$\text{banana}$</th>
<th>$\text{banana}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>$\text{banana}$</td>
<td>$\text{banana}$</td>
</tr>
<tr>
<td>3</td>
<td>$\text{anana}$</td>
<td>$\text{anana}$</td>
</tr>
<tr>
<td>6</td>
<td>$\text{nana}$</td>
<td>$\text{nana}$</td>
</tr>
<tr>
<td>2</td>
<td>$\text{ana}$</td>
<td>$\text{ana}$</td>
</tr>
<tr>
<td>5</td>
<td>$\text{na}$</td>
<td>$\text{na}$</td>
</tr>
<tr>
<td>1</td>
<td>$a$</td>
<td>$a$</td>
</tr>
</tbody>
</table>
trie

- rooted tree, edges labeled by characters
- node: concatenation of the edge labels on the path from the root to the node
- trie for a set of strings: minimal trie whose nodes represent all strings in the set
- set is prefix free $\Rightarrow$ nodes representing strings are leaves
- compact trie: replace branchless path with a single edge (concatenation of the replaced edge labels)
trie, $\mathcal{T} = \{operation, research, reservation, result\}$
compact trie, \( \mathcal{T} = \{\text{operation, research, reservation, result}\} \)
suffix tree $ST_T$

- Compact trie of the set of suffixes of $T$
- $O(N)$ nodes, constructed in linear time
- Sentinel character $\$$ to make the set of suffixes prefix free
- Walking down the path: $O(|P|)$
- Searching the subtree: $O(R)$
- Insertion/deletion of a string $S$ in $O(|S|)$ (needs suffix links)
- Suffix link: pointer from a node representing the string $a\alpha$ ($a \in \Sigma$, $\alpha \in \Sigma^*$) to a node representing $\alpha$
suffix tree $ST_{\mathcal{T}}$ for $\mathcal{T} = \{\text{banana}\}$
External Memory Techniques

- Pat Trees
- String B-Trees
- Self-adjusting Skip List
External Memory Techniques: Pat Trees

- Patricia tries
  - related to compact trie
  - edge labels contain only the first character (branching character) and the length of the corresponding compact trie label (skip value)
  - delay access to the text as long as possible

- Pat Tree $PT_T$
  - Patricia trie for the set of suffixes of a text $T$
  - String matching with pattern $P$, $O(|P| + R)$
    - only the first character of each edge is compared to the corresponding character in $P$, skip value tells how many characters are skipped
    - success: all strings in the resulting subtree have the same prefix of length $|P|$ ($\Rightarrow$ all of them or none have prefix $P$)
Patricia trie, $\mathcal{T} = \{\text{operation, research, reservation, result}\}$
Pat tree \( PT_T \) for \( T = \{ \text{banana}\} \)
External Memory Techniques: Pat Trees [contd.]

- binary encoding of the characters
  - every internal node has degree two
  - no need to store the first bit of the edge label (left/right distinction encodes already)

- lexicographic naming of a set $S$ of strings, lexicographic order $\leq_L$
  - $n : S \to \mathbb{N}, s \mapsto n(s)$
  - $\forall s_i, s_j \in S$
    - $n(s_i) = n(s_j) \iff s_i = s_j$
    - $s_i \leq_L s_j \iff n(s_i) \leq n(s_j)$
  - arbitrary long strings can be compared in constant time
  - construct lexicographic naming: sort $S$ and use the rank of $s_i$ as name $n(s_i)$

- store only suffixes at the beginning of a word
Compact Pat Tree $CPT_T$ (Clark and Munro)

- efficient for searching static text in primary storage
- partition the Pat Tree into pieces that fit into a disk block, offset pointers point to a suffix in the text or to a subtree (partition)
- little more storage ($\geq \log_2 N$ bits per suffix), size $3.5 + \log_2 N + \log_2 \log_2 N + O\left(\frac{\log_2 \log_2 N}{\log_2 N}\right)$ bits per node
- compact tree encoding (string $\rightarrow$ binary)
- large skip values are unlikely (fixed number of bits reserved to hold the skip value: $\log_2 \log_2 \log_2 N$) if large skip value (overflow) insert another node and distribute skip bits
- searching: $O(\text{SCAN}(|P| + R) + \text{SEARCH}(N))$ I/Os
- path from root to leaf: at most $1 + \left\lceil \frac{H}{\sqrt{B}} \right\rceil + \left\lceil 2 \cdot \log_B N \right\rceil$ pages (height $H$, $O(\sqrt{B} \cdot \log_B N)$, worst: $\Theta(N)$)
External Memory Techniques: String B-Trees (Ferrapina, Grossi)

- **Time, Space**
  - string matching (pattern $P$) in $\mathcal{O}(\text{Scan}(|P| + R) + \text{Search}(N))$ I/Os
  - insert/delete string $S$ in $\mathcal{O}(|S| \cdot \text{Search}(N + |S|))$ I/Os
  - space requirement: $\Theta\left(\frac{N}{B}\right)$ blocks
  - Construction by insertion: $\mathcal{O}(N \cdot \text{Search}(N))$ I/Os
  - best performance per operation in worst-case

- **Structure**
  - combination of B-Trees and Patricia tries
  - keys are stored at the leaves (logical pointers to the strings stored in external memory), internal nodes contain copies of some of these keys
  - node $v$ stored in a disk block, contains an ordered string set $S_v \subseteq S$, (leftmost/rightmost string: $L(v)/R(v)$)
  - B-Tree property: $b \leq |S_v| \leq 2 \cdot b \ (b = \Theta(B))$
as you can see this is a string data structure

1 4 8 12 16 21 24 26 33 38
External Memory Techniques: String B-Trees [contd.]

Search procedure

- Standard B-tree performs a branch at every node $\rightarrow$ read part of the string to compare with (takes too long)
- Optimization: use a Patricia trie to read only few characters $\rightarrow$ problem: start reading pattern $P$ from the beginning at every level
- Solution: use parameter $lcp$ (longest common prefix) to determine, how many characters are ok
Insertion and deletion

- Insertion of an item into a B-tree means searching its position and then inserting (perhaps some splits occur)
- Insertion of a string $S$ means inserting all its suffixes (insert $|S|$ strings)
- $succ$ Pointers: any suffix $S_i[j, |S_i|]$ of string $S_i$ has a pointer to the next suffix $S_i[j + 1, |S_i|]$
- any string in the B-tree shares its first few characters with one of its adjacent strings
- insert the longest suffix (the string itself) and use the $succ$ Pointer of its neighbour to insert the next suffix
- Attention: rebalancing (split, merge) needs to update the $succ$ Pointers as well
External Memory Techniques: Sorting Strings

Sorting Strings in External Memory is not nearly as simple as it is in Internal Memory

- Use a String B-tree to sort $K$ strings: $O(K \cdot \log_B K + \frac{N}{B})$
- Doubling Algorithm (Karp, Miller, Rosenberg): $O(\text{SORT}(N) \cdot \log_2 N)$ I/Os (also used for suffix array construction)

<table>
<thead>
<tr>
<th>4</th>
<th>b</th>
<th>4</th>
<th>ba</th>
<th>4</th>
<th>bana</th>
<th>4</th>
<th>banana$$</th>
<th>4</th>
<th>bananan$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>2</td>
<td>an</td>
<td>3</td>
<td>anan</td>
<td>5</td>
<td>na$$</td>
<td>2</td>
<td>ana$$</td>
</tr>
<tr>
<td>5</td>
<td>n</td>
<td>5</td>
<td>na</td>
<td>6</td>
<td>nana</td>
<td>5</td>
<td>na$$</td>
<td>5</td>
<td>na$$</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
<td>a$</td>
<td>1</td>
<td>a$$</td>
<td>1</td>
<td>a$$</td>
<td>1</td>
<td>a$$</td>
</tr>
</tbody>
</table>

Full text indexing, Christian Sommer, WS 04/05
External Memory Techniques: Self-adjusting structures

- Repetition: Splay trees (Tarjan)
  - move accessed node to the root (MTF strategy)
  - Static Optimality Theorem
  - amortized analysis

- Repetition: Skip lists (Pugh)
  - randomized data structure, tree-approximation
  - every item has several pointers to its successors
  - pointers on level $i$ form a doubly linked list $L_i$
  - internal skip list:
    - probability to add another level on an item: $\frac{1}{2}$ (internal)
    - $E[h] = \log_2 n$ ($h$ is the maximum level), $E[|L_i|] = \Theta(2^{h-i})$
    - search, insert, delete: $O(\log_2 n)$
  - external: probability: $\Theta\left(\frac{1}{B}\right)$ (Callahan), height: $O(\log_B n)$
Biased skip list (Ergu)

- MTF strategy: every item has a move to front rank \( r \) (MTF-rank) (small rank ⇔ high level in skip list)
- search, insert, delete: \( \mathcal{O}(\log_2 r) \)
- on a query:
  * promote accessed item to the top levels, set rank to 1
  * demote \( \Theta(\log_2 r) \) items to lower levels
  * increment the MTF-ranks of all items with rank smaller than \( r \)
- selecting the demoted elements: chosen by a Random Walk with weights computed by counters stored in each item (approximately LRU (least recently used) strategy)
Self-adjusting skip lists (SASL)

- randomized structure, frequent items get to remain at the highest levels of the skip list
- problem of splay trees: string as atomic item (hash) doesn’t solve searching (partial match queries), dictionary doesn’t fit into the main memory
- $K$ Strings $S_1 \ldots S_K$, $\sum |S_i| = N$
- sequence of $m$ String searches $S_{i1} \ldots S_{im}$, $n_i$: number of times $S_i$ is queried: $\mathcal{O}(\sum_{j=1}^{m} \frac{S_{ij}}{B} + \sum_{i=1}^{K} n_i \log_B \frac{m}{n_i})$
- insertion, deletion of $S$: $\mathcal{O}(\frac{|S|}{B} + \log_B K)$
- space requirements: $\mathcal{O}(\frac{N}{B})$ disk pages
Literature

- Algorithms for Memory Hierarchies: Advanced Lectures
  - Full-Text Indexes in External Memory (Juha Kärkkäinen, S. Srinivasa Rao)

- other papers and books
  - Self-adjusting Data Structures for External Memory String Access (V. Ciriani, P. Ferragina, F. Luccio, S. Muthukrishnan)
  - The String B-Tree: A New Data Structure for String Search in External Memory and Its Applications (P. Ferragina, R. Grossi)
  - Algorithmen und Datenstrukturen, 4. Auflage, Skip-Liste p.42 (T. Ottmann, P. Widmayer)
  - Efficient External-Memory Data Structures and Applications (L. Arge)
  - On Sorting Strings in External Memory (L. Arge, P. Ferragina, R. Grossi, J.S. Vitter)