Contents

1 Inverted Files (Cont.)

1.1 Concerns and Approaches for Inverted Indexing

- Insertion overhead depends on the frequency of update, requirement on the currency of the index, and workload of the system.
- For library applications, insertion overhead is not a problem; for newspaper and World Wide Web indexing, it is.
- Factors to consider: size of main memory, temporary disk space, and batched or online update of the index.

1.2 Batched Insertion by Sorting

1. Collect all new documents.
2. Extract the terms for each document and prepare inverted file:

term	rec#
pap	1
report	1
novel	1
novel	1
...	..
report	2
human	2
...	..

$\xrightarrow{sort}$

term	rec#
human	2
novel	1
novel	1
pap	1
report	1
report	2
...	..

$\xrightarrow{unique}$

term	rec#	freq
human	2	1
novel	1	2
pap	1	1
report	1	1
report	2	1
...	..	

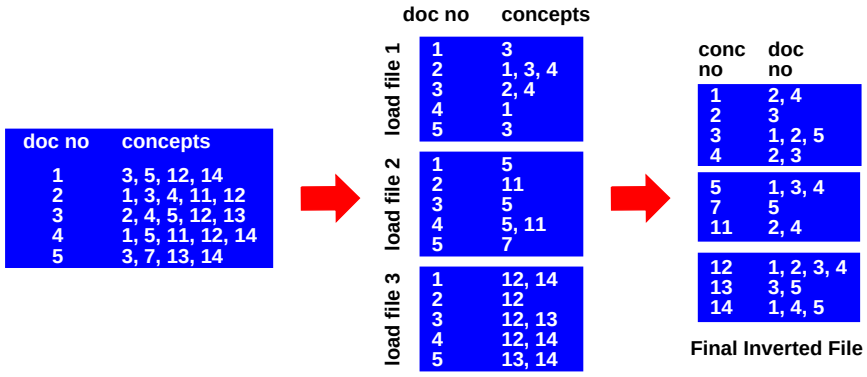
1.3 Batched Insertion Using a Temporary Binary Tree

- Sorting requires a lot of main memory and/or temporary disk space.
- The solution is to parse the documents and insert the terms into a dynamic file structure (Binary Search Tree) right away. Then, terms are retrieved from the tree in sorted order and build the final inverted file.
- Slow because of insertions on the binary tree.
- Note that all of these techniques are based on the assumption that postings lists are stored in a single sequential file to save storage. It may be a bad idea from update point of view.

1.4 Fast Inversion Algorithm

- The goal is fast insertion of *large* number of documents utilizing large main memory available in today's computers.

1.4.1 Example of Fast-Inv



Distribution of concepts in load files	load file 1	2	3
no of concepts	4	3	3
no of doc/concept pairs	8	6	9

1.4.2 Algorithm of Fast-Inv

1. Given the document vector file, find out the number of document-concept pairs in the documents.
2. Given the memory size, try to divide the document vector file into j load files so that:
   - each load file, containing document-concept pairs, can be loaded into the main memory
   - each load file has about the same number of unique concepts (terms) in it
   - j is as small as possible.
   - concept numbers in load file i are greater than concept numbers in load file j, for all $j < i$.

3. Build the inverted file starting from the first load file.

- Because of the way load files are created, the inverted file is build starting from the lowest concept number in ascending order.
- The information collected about the documents and concepts allows storage to be preallocated for the posting file. Postings can be inserted directly into the file without searching or causing storage to be allocated (as in the case of linked list).

2 Full-Text Scanning (Pattern Matching)

- Indexing causes too much storage and processing overhead to complex search (e.g., regular expressions and proximity search)
- Post-processing (list merge and intersection) becomes more and more expensive
- Inverted indexes are good for relatively static environment.

- Full-text scanning has no storage and processing overhead
- Good for dynamic environment
- Search is done on the raw text — all information in the text is theoretically searchable
- Slow but acceptable when file size is small (as is typically the case for files on UNIX)
- Compromise: A coarse index followed by localized search

2.1 Pattern Matching Algorithms

2.2 Brute-Force Method

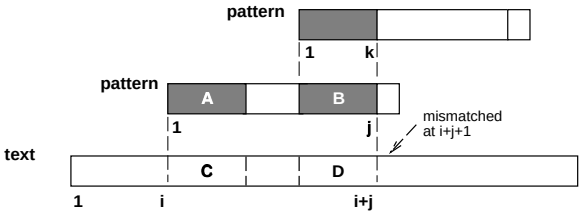
- Assume no knowledge on the pattern.
- n = text length; m = pattern length.
- Performance (try it out):

		text	pattern	comparisons
Worst case	$(n-m+1)\times m$	aaaaaaa	aab	15
Best case	$n-m+1$	abcdefg	xab	5
Between		abababa	abc	11

2.3 Knuth-Morris-Pratt (KMP) Method

- Preprocessing of pattern: $O(m)$.
- Performance depends on number of repeating sub-patterns within the pattern:
  pattern = abcdefg performs much better than pattern = abababc
- Upon a mismatch, the brute-force method shifts 1 character a time whereas KMP may shift more than 1 character a time.
- Given pattern=abaab, we can analyze the pattern and precompute the number of character positions to shift when a mismatch at a certain position occurs.

first mismatched pattern position	number of shifts	example
1 [a]	1	...X..... abaab# abaab#
2 [b]	1	...aX..... abaab# abaab#
3 [a]	2	...abX..... abaab# abaab#
4 [a]	2	...abaX..... abaab# abaab#
5 [b]	3	...abaaX..... abaab# abaab#
6 [#]	3	...abaabX..... abaab# abaab#



- To find the maximal matching prefix and suffix from the sub-pattern that has been matched so far (shaded parts).
- Define the end of the maximal matching prefix and suffix (positions k and j) as the *borders* and the mismatch position $(i+j+1)$ as the *hot* position.
- From the pattern, we know that $A = B$. If the text matches the pattern upto the j^{th} pattern character, we know that $C = D = A = B$. Thus, we can shift the pattern so that A and D align and restart matching at position $i+j+1$.

- If there is no repeating sub-pattern within pattern [1..j], then the pattern can be shifted by j characters.
- Patterns with repetitions take more time to match in general. Consider the pattern `aaaaaaa`.
- Performance: $2 \times n$.
- This version of KMP (as used in **[S]**, p. 261) doesn't examine the mismatched pattern character.
- The algorithm can be further improved by:
  1. Shift the same number of characters as before.
  2. If the pair of characters following the two borders are the same, continue recursively as if a mismatch occurs immediately after the shift.
  3. Stop step 2 when the pattern character under the hot position is different from the original one, or the pattern has moved to the right of the hot position.
- Note that when the pattern character at the hot position is the first pattern character, the maximal matching prefix and suffix are null and the borders are defined as at the left of the first pattern character.
- When no matching prefix and suffix can be found, the prefix border is at the left of the first pattern character and the suffix border is at the left of the hot position.
- It is also assumed that *we don't know what the mismatching text character is* (ie., we only know whether it *matches* or *doesn't* match the pattern character).
- The new shifts are:

```
1 2 3 4 5 6
-----------
a b a a b #
1 1 2 2 3 3  <- original shifts
1 1 3 2 4 3  <- new shifts
    ^ ^ ^ ^
    | | | |_ same as previous method
    | | |__________ pat[2] = pat[5]: shift 1 more pos
    | |               pat[5] != pat[5], stop
    | |____________ pat[2] != pat[4]
    |
    a b a a b #     pat[1] = pat[3]: shift 1 more pos,
      a b a a b #      pattern shifted beyond the original
                       mismatched pattern char, stop
```

- Another example:

```
         1 2 3 4 5 6 7
         -------------
         a b a a b a #
shifts   1 1 3 2 4 6 3
                 ^
                 |_ mismatch:         shift 3 characters
               a b a a b a #
                 ^
                 |_ mismatched again: shift 2 characters
                 a b a a b a #
                 ^
                 |_ mismatched again: shift 1 characters
                    total shifted:          6 characters
```

- In a real implementation (cf Chap 10 of text), pointers are set up pointing to the text and pattern string. Therefore, instead of "shifting" a pattern, pointers are advanced. Notice that:
  - The text pointer is either stationary or incremented by one, whereas the pattern pointer is set according to the pattern table next[].
  - The pattern table next[] points to the pattern character to be matched next. It is different from the shift table in the first example (from Salton Chapter 8.6.3).

- Further example: basic versus improved shift array

```
                     1 1  1
   1 2 3 4 5 6 7 8 9 0 1  2
   -----------------------
   a b r a c a d a b r a  #
   1 1 2 3 3 5 5 7 7 7 7  7  <- original shifts
   1 1 2 4 3 6 5 8 8 9 11 7  <- new shifts
         ^ ^ ^     ^   ^
         | | |     |   |_ pat[4]=pat[11] and pat[1]=pat[4]
         | | |     |_ 7 becomes 8, pat[2] = pat[9]
         | | |_ 5 becomes 6, pat[1] = pat[6]
         | |_ no change, pat[2] != pat[5]
         |_ 3 becomes 4, pat[1] = pat[4]
```

- Using next[]:

```
                        1 1  1
         1 2 3 4 5 6 7 8 9 0 1  2
         -----------------------
         a b r a c a d a b r a  #
next[]   0 1 1 0 2 0 2 0 1 1 0  5
```

- next[]=0 means advance text pointer and reset pattern pointer to 1, which means shifting the pattern to the right of the mismatching pattern character. E.g., next[4]=0 is equivalent to shifting 4 characters.
- Question: can you establish a relationship between the shift array and the next array?

2.4 Hardware Pattern Matchers

- Associative memory
- VLSI cellular array
- Hardware implementation of Finite State Automaton
- Too expensive
- Loading data from disk into pattern matcher is the main bottleneck
- Clever indexing yields much better profits