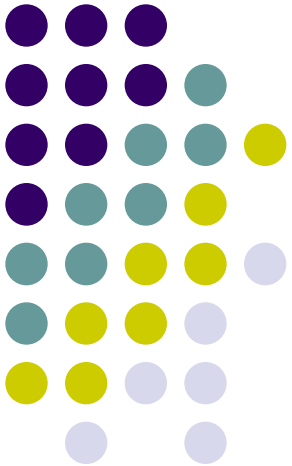
Positional Data Organization and Compression in Web Inverted Indexes

Leonidas Akritidis
Panayiotis Bozanis

Department of Computer & Communication Engineering, University of Thessaly, Greece

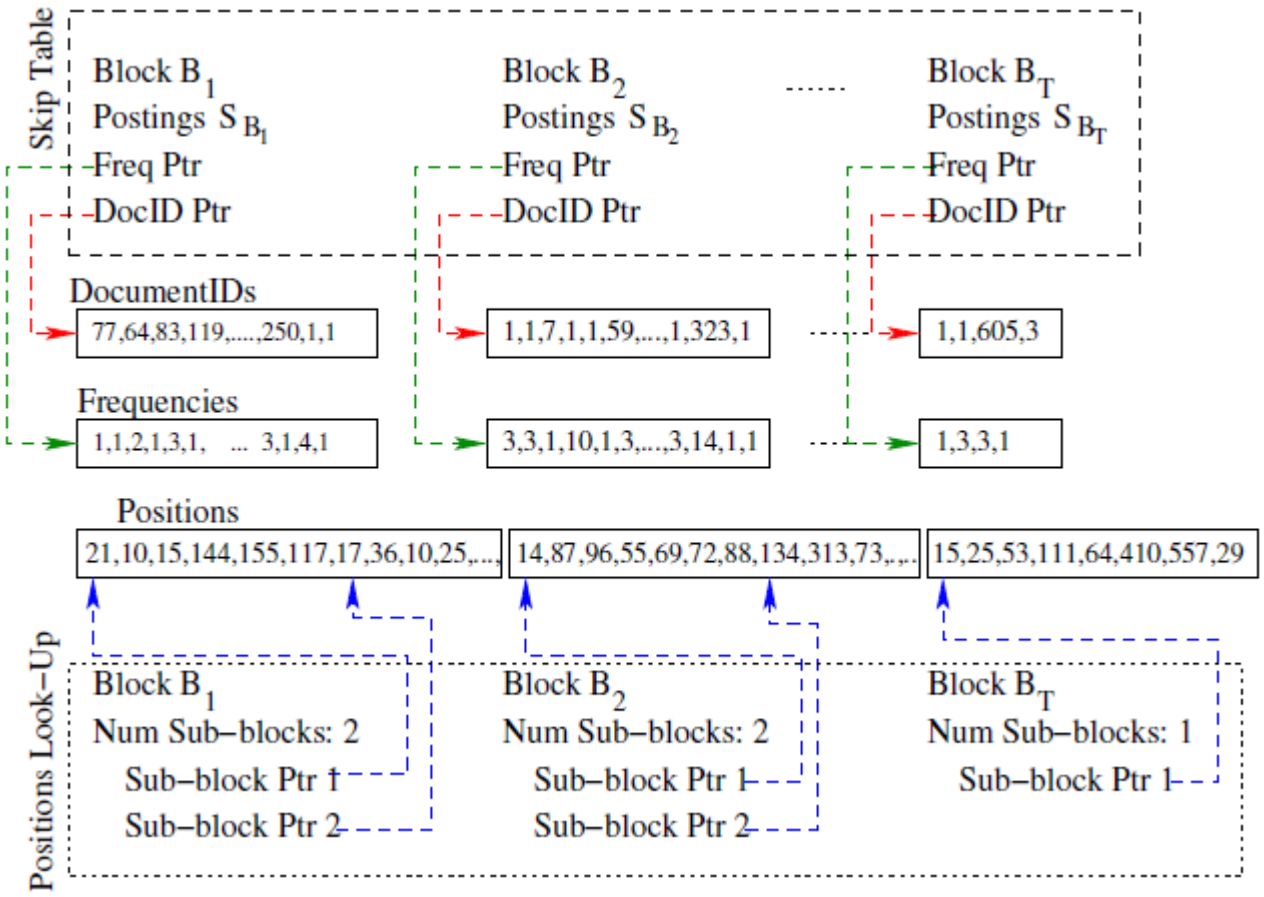


23rd International Conference on Database and Expert Systems Applications
DEXA 2012, September 3-7, Vienna, Austria

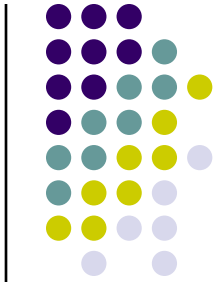
Introduction

- We present a method for the organization and compression of the positional data in Web Inverted Indexes.
- Important problem since the positions enable:
  - Evaluation of ranked proximity queries.
  - Evaluation of exact phrase queries.

Block-based index setup



Block-based index setup

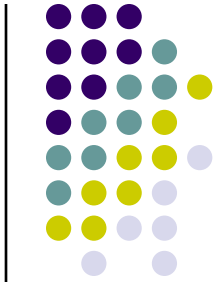


- It allows partial decompression of the docID blocks, without touching the rest of the index data (frequencies, positions, etc).
- Useful during DAAT/TAAT query processing where we are interested in locating docIDs smaller than or equal to a given value.
- It requires:
  - A skip table to jump forward in the inverted list.
  - A positions look-up structure (more later).

Inverted index compression

- Many algorithms, divided in two categories:
- Individual integer compression methods
  - Applicable to single integer values.
  - Variable Byte, Elias Gamma/Delta, Golomb/Rice.
- Group integer compression methods
  - Applicable to entire bundles of integers.
  - VSEncoding, PForDelta (P4D), Optimized P4D.
  - Fast, but require decoding the entire block.

Positional data

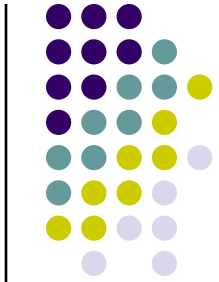


- The group encoding methods are ideal for docIDs
- But what about the positional data?
- The positional data dominate the index: Each posting includes on average 2.5-3.5 positional values.
- Hence, decoding *all* the positions for *all* of the candidate postings is very expensive.

Positional data access during query processing

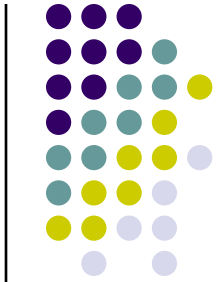
- Query processing optimization.
- Divide the processing in two stages:
  - On the first step, quickly identify the most suitable results by using docIDs and frequencies only.
  - On the second step, perform a more refined processing by accessing the positional data for the *K* best results of the previous phase only.
- Apparently, we are only interested in accessing the positions of specific postings.
- Decoding entire blocks is redundant.

Positional Data Organization



- Consequently, the group encoding methods are rendered inappropriate.
- Moreover, we do not know exactly where the positional values of a posting are stored.
- For this reason, two strategies have been proposed:
  - Create a tree-based positions look-up structure.
  - Construct indexed lists.

Positional data Look-ups

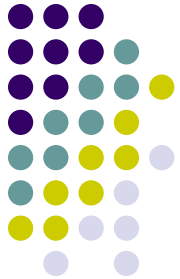


- The look-up structure requires one look-up per posting, thus decelerating processing.
- The indexed list requires one pointer per posting pointing to the respective positional data, thus it is prohibitively expensive.

PFBC: Positions Fixed-Bit Compression

- PFBC is designed to overcome all these problems.
- For each inverted list block, it employs a fixed number of bits to binary-encode each positional value.
- Then, by using the frequency values, we are able to *calculate* the exact location of the positions for each posting, *without look-ups*.
- We decode *only* the data *actually* required.

PFBC: Index organization



- PFBC is compatible to all state-of-the-art inverted list partitioning approaches.
  - Compressed Embedded Skip Lists, Skips every 128 postings, Skips every $\sqrt{F_t}$ postings (F_t: number of documents containing t), etc.
- We store all the positional values contiguously.
- For each inverted list block B_i we store:
  - A pointer R_{Bi} pointing to the beginning of the positional data of the postings of the block B_i.
  - A C_{Bi} value representing the number of bits used to encode the positions of the block B_i.

PFBC: Index organization

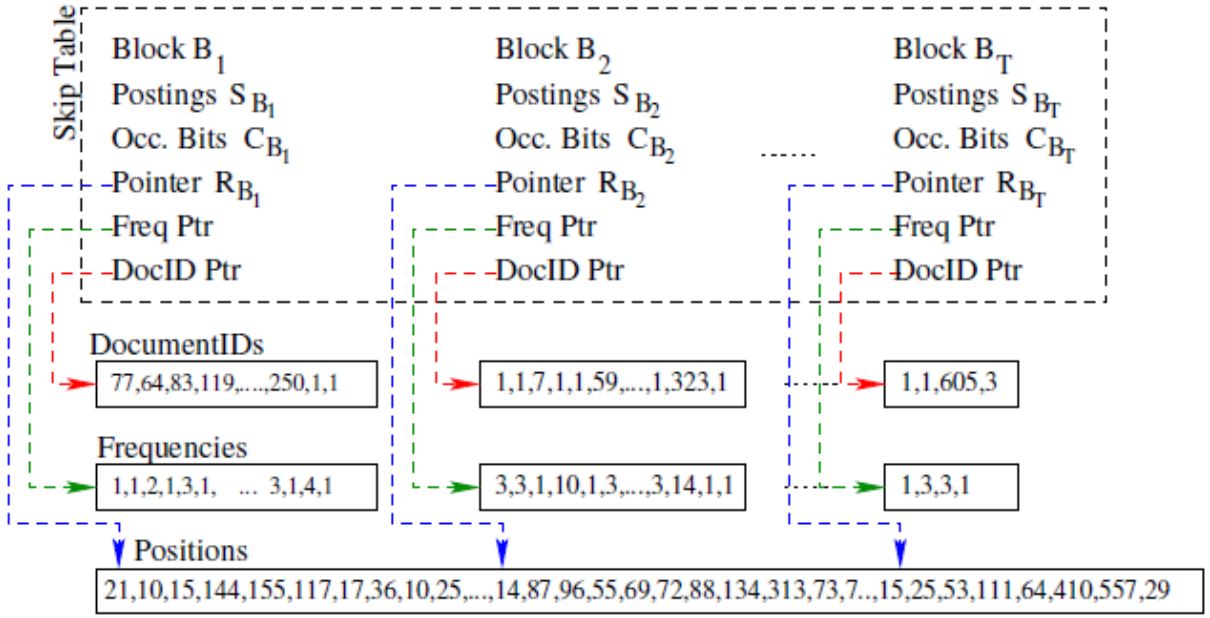
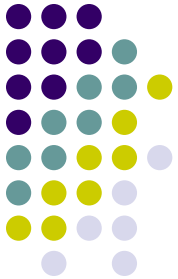


Fig. 1. Organizing an inverted list into blocks according to PFBC (lower part) and the auxiliary skip table (upper part). Notice that the positional data is packed separately at the end of the inverted list. For each block B_i of the list, we store within the skip table (a) a pointer R_{B_i} pointing at the beginning of the corresponding compressed positions and (b) a value C_{B_i} denoting the number of bits we used to encode the positions of the block.

PFBC: Compression Phase

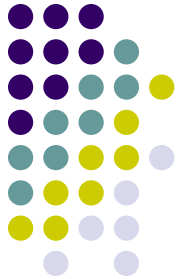


byte $PFBC - Encode(K, p[K])$

1. **int** $i \leftarrow 0, p_{max} \leftarrow 0, C \leftarrow 0$
2. **byte** $\mathcal{P}$
3. **while** $(i < K)$ {
4. **if** $(p_i > P_{max})$ {
5. $p_{max} = p_i$
6. }
7. $i++$
8. }
9. $C \leftarrow \lceil \log_2(p_{max} - 1) \rceil$
10. $\mathcal{P} \leftarrow$ **allocate** $\lceil KC/8 \rceil$ bytes
11. $i \leftarrow 0$
12. **while** $(i < K)$ {
13. $write(p_i, \mathcal{P}, C)$
14. $i++$
15. }
16. **return** $\mathcal{P}$

- Encode a bundle of *K* positional values.
- **Steps 3-8:** Identify the largest positional value p_{max}.
- **Step 9**: Calculate *C*, i.e. the number of bits required to represent all *K* values.
- **Step 13**: The function *write()* stores an integer into a compressed sequence *P* by using *C* bits.

PFBC: Decompression Phase



```
int PFBC − Decode(j, B_i, P)
1.  int x ← 0, s ← 0
2.  while (x < j) {
3.      s ← s + f_{x,B_i}
4.      x++
5.  }
6.  int start ← R_{B_i} + sC_{B_i}
7.  x ← 0
8.  while (x < f_{j,B_i}) {
9.      p[x] ← read(P, C_{B_i}, start)
10.     start ← start + C_{B_i}
11.     x++
12. }
13. return p
```

$$\mathcal{S}_j = R_{B_i} + C_{B_i} \sum_{x=0}^{j-1} f_{x,B_i}$$

- Decode the positional data of posting *j*, of the block B_i.
- **Steps 2-5:** Sum up all the frequency values of the previous postings of B_i.
- **Step 6:** Locate the bit where the positions of *j* start from.
- **Step 9:** The *read()* function reads each encoded position from the *P* vector.

PFBC Gains

- It facilitates direct access to the positional data without requiring expensive look-ups. Therefore, query processing is accelerated.
- It saves the space cost of maintaining a separate look-up structure; the required data is stored within the skip table.
- It needs much fewer pointers than the indexed lists.
- It enables decoding of the information actually needed, without decompressing entire blocks of integers.

Experiments

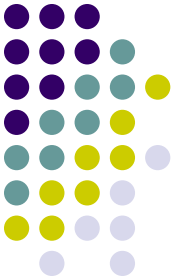
- Compression experiments: PFBC against OptP4D and VSEncoding.
- Organization experiments: PFBC against the tree look-up structure of [Yan et.al] and the indexed list of [Transier and Sanders].
- Document collection: ClueWeb-09B dataset (~50 million documents, ~1.5 TB).

Experiments: Index size

Data Structure	OptP4D	VSEncoding	PFBC
Inverted Index	90.8	90.2	92.0
Skip Table	1.7	1.7	1.7
Pointers to positions	-	-	2.1
Positions look-up	4.1	4.1	-
Total	96.6	96.0	95.8

- PFBC introduces a slight loss in the overall index size (increase of 1%-2%).
- This loss is amortized by the reduced size of the accompanying data structures (skip table, pointers to positions).

Experiments: Query throughput



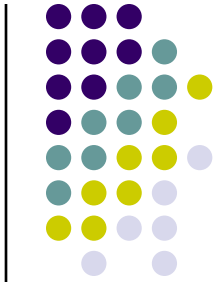
K		OptP4D	VSEncoding	PFBC
$K = 200$	Decompressed positions	2,756,128	2,756,128	374,251
	Decompressed positions/query	55,123	55,123	7,485
	Total access time (msec)	0.11	0.11	0
	Total decompression time (msec)	5.56	5.14	1.01
	Average time per query (msec)	0.11	0.10	0.02
$K = 1000$	Decompressed positions	11,192,608	11,192,608	1,044,234
	Decompressed positions/query	223,852	223,852	20,885
	Total access time (msec)	0.31	0.31	0
	Total decompression time (msec)	23.94	21.10	4.02
	Average time per query (msec)	0.48	0.42	0.08

- Recall: The query processing is performed in two phases; the positions are retrieved for the *K* best results of the first phase only.
- PFBC touches much fewer data.
- PFBC is look-up free.
- PFBC offers ~5 times faster decompression.

Conclusions

- We presented PFBC, a method for effective organization and compression of the positional data in Web inverted indexes.
- PFBC uses a fixed number of bits to encode the positions of an inverted list block.
- The slight increase in the overall index size is amortized by the slight decrease in the assistant data structures sizes.
- Much faster decompression.
- No look-ups for the positions are required.
- Allows us to decode the data *actually* needed.

The end



Thank you for watching!

Any questions?