Index Structures and Query Execution

- TAAT and DAAT query execution techniques
- index compression
- putting it all together:

  blockwise compression, DAAT, and list caching

Query Execution: TAAT vs. DAAT

- How can we efficiently execute queries using an inverted index?
- *term-at-a-time (TAAT)* vs. *document-at-a-time (DAAT)* query processing
- Assume the following data structures:

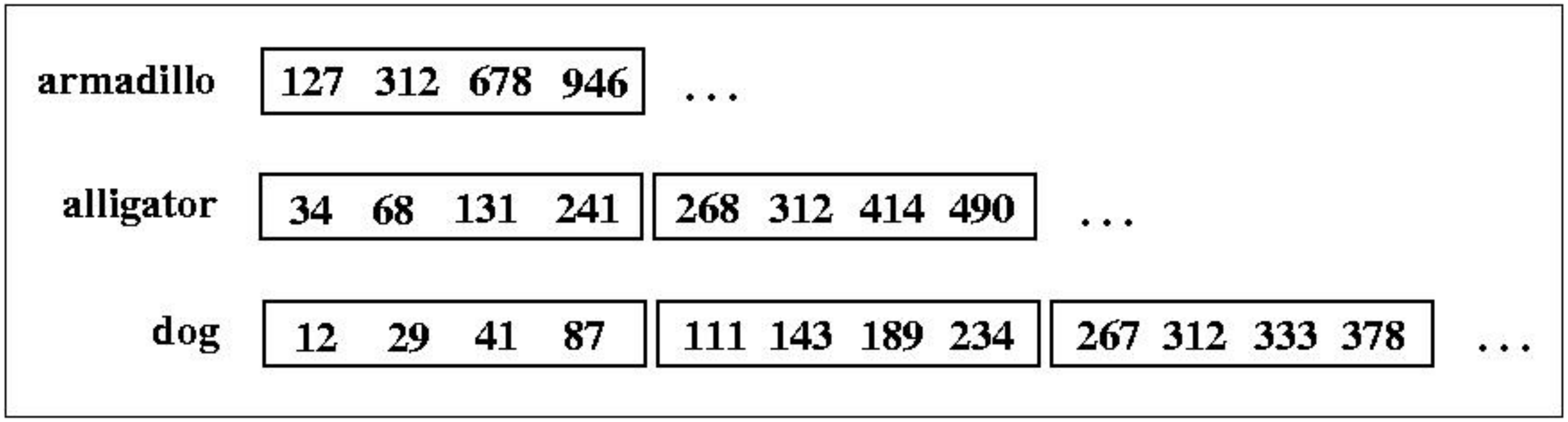
(1) An inverted index with an inverted list for each distinct word. Each posting contains the docID and (not shown) the number of occurrences of the word in the document, and their positions and contexts.

(2) A hash table storing for each distinct word *t* in the data set, a pointer to the start of its inverted list, and f_t, the number of docs containing *t*.

(3) A table containing for each docID, the URL of the page, the length of the page |*d*|, and maybe also the pagerank value of the page if needed.

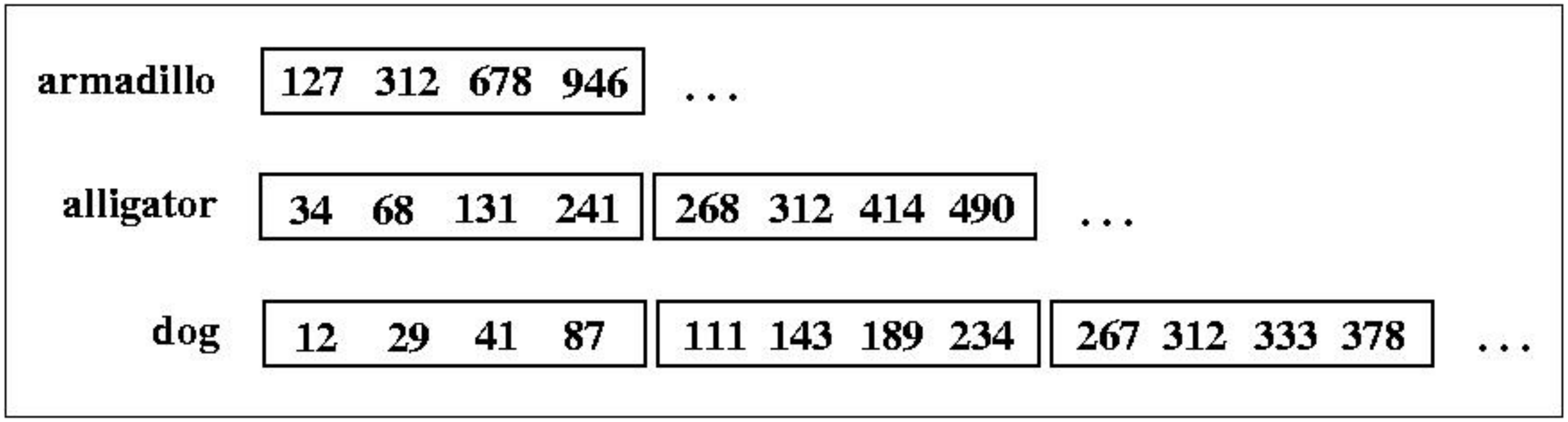
aardvark	3452, 11437, …..
⋮	
arm	4, 19, 29, 98, 143, ...
armada	145, 457, 789, ...
armadillo	678, 2134, 3970, ...
armani	90, 256, 372, 511, ...
⋮	
zebra	602, 1189, 3209, ...

Term-At-A-Time Query Processing



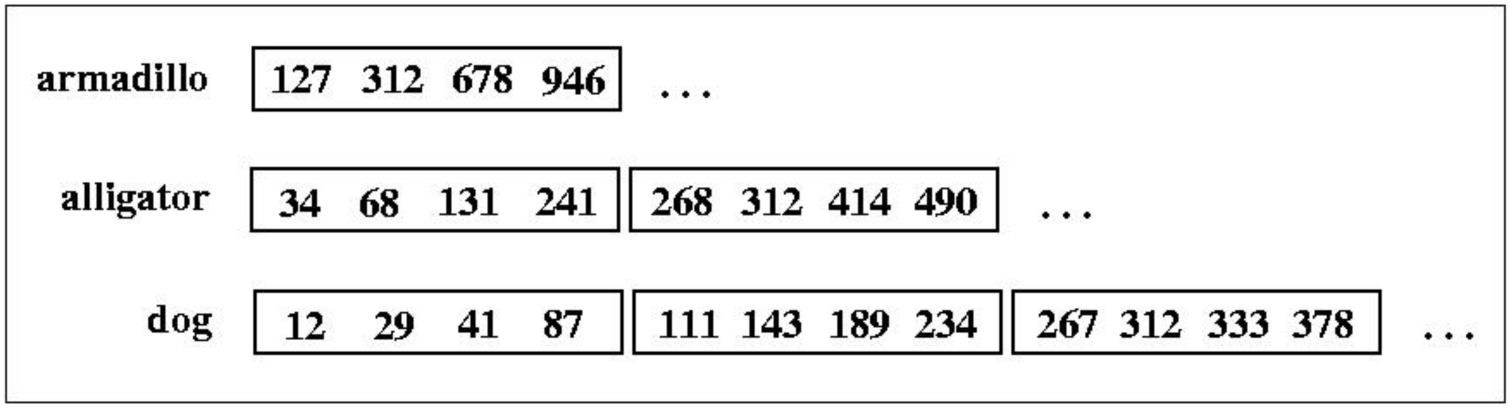
- process one inverted list at a time, from shortest to longest list
- assume AND semantics: *"all query words must occur in result"*
- initialize an empty hash table
- for each posting in shortest list, create an hash entry with the docID as key and the cosine contribution for this term as value
- for each posting in the other lists, check if there is an entry in the hash table; if yes, add term contribution to cosine
- output results with ten highest values that contained all terms

Document-At-A-Time Query Processing



- assume again AND semantics
- all inverted lists in the query are traversed simultaneously from left to right
- can be implemented using four simple primitives:
  - *openList(t)* and *closeList(lp)* open and close the inverted list for term *t* for reading
  - *nextGEQ(lp, k)* find the next posting in list *lp* with *docID* >= *k* and return its docID. Return value > MAXDID if none exists.
  - *getFreq(lp)* get the frequency of the *current* posting in list *lp*
- inverted list access: similar to opening a file or a socket, or to using a cursor in a databases

Document-At-A-Time Query Processing



- open all 3 inverted lists for reading using *open()*
- this returns 3 pointers *lp0*, *lp1*, and *lp2* to the starts of the lists
- call *d0 = nextGEQ(lp0, 0)* to get docID of first posting in *lp0*
- call *d1 = nextGEQ(lp1, d0)* to check for matching docID in *lp1*
- if *(d1 > d0),* start again at first list and call *d0 = nextGEQ(lp0, d1)*
- if *(d1 = d0),* call *d2 = nextGEQ(lp2, d0)* to see if *d0* also in *lp2*
- if *(d2 > d0),* start again at first list and call *d0 = nextGEQ(lp0, d2)*
- if *(d2 = d0),* then *d0* is in all three lists; compute its score; then continue at first list and call *d0 = nextGEQ(lp0, d2+1)*
- whenever a score is computed for a docID, check if it should be inserted into heap of current top-10 results; at the end, return results in heap

```
for (i = 0; i < num; i++)  lp[i] = openList(q[i]);

did = 0;
while (did <= maxdocID)
{
  /* get next post from shortest list */
  did = nextGEQ(lp[0], did);

  /* see if you find entries with same docID in other lists */
  for (i=1; (i<num) && ((d=nextGEQ(lp[i], did)) == did); i++);

  if (d > did)  did = d;      /* not in intersection */
  else
  {
    /* docID is in intersection; now get all frequencies */
    for (i=0; i<num; i++)  f[i] = getFreq(lp[i], did);

    /* compute BM25 score from frequencies and other data */
    <details omitted>
    did++;    /* and increase did to search for next post */
  }
}

for (i = 0; i < num; i++)  closeList(lp[i]);
```

Remember: Ranking Functions

- one example of the *Cosine Measure*

$$
\begin{aligned}
F(d, t_0, \ldots, t_{m-1}) &= \sum_{i=0}^{m-1} \frac{w(q, t_i) \cdot w(d, t_i)}{\sqrt{|d|}}, \\
w(q, t) &= \ln(1 + N / f_t), \text{and} \\
w(d, t) &= 1 + \ln f_{d,t},
\end{aligned}
$$

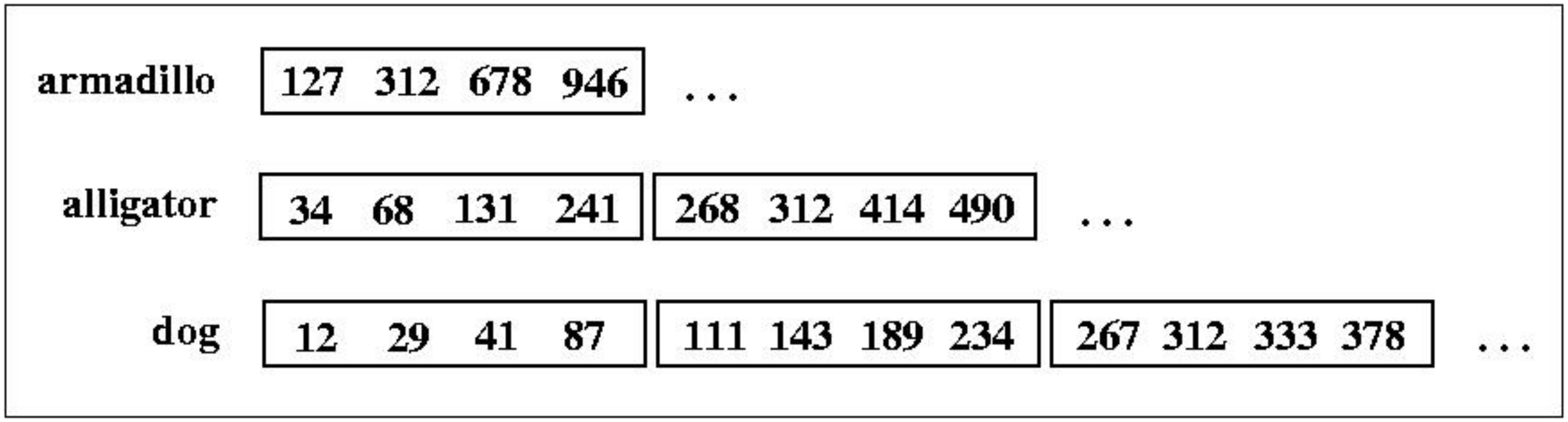
- another popular function: BM25

$$BM25(q,d) = \sum_{t \in q} \log\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right) \times \frac{(k_1 + 1) f_{d,t}}{K + f_{d,t}}$$

- ***N***: total number of documents in the collection;
- ***f**$_t$*: number of documents that contain term ***t***;
- ***f**$_{d,t}$*: frequency of term ***t*** in document ***d***;
- **|*d*|**: length of document ***d***;
- **|*d*|$_{avg}$**: the average length of documents in the collection;
- ***k**$_1$* and ***b***: constants, usually ***k**$_1$* = 1.2 and ***b*** = 0.75

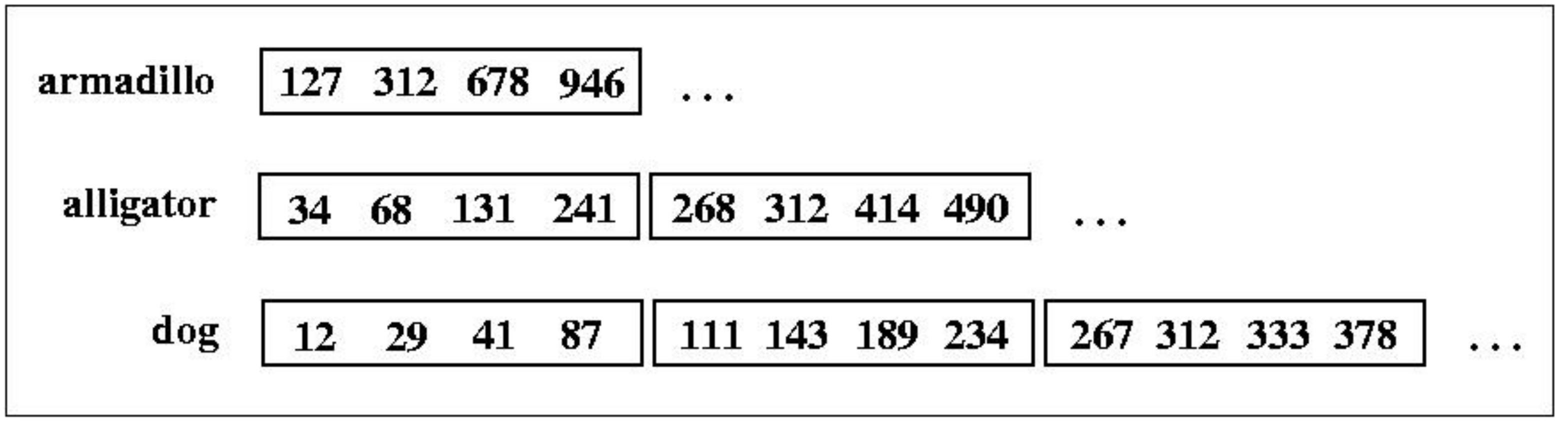
$$K = k_1 \times \left((1 - b) + b \times \frac{|d|}{|d|_{avg}}\right)$$

Document-At-A-Time Query Processing



- **DAAT uses no extra space for hash table (only heap for top-10)**
- **faster and better than TAAT for search engines**
- **very simple to implement - just a few lines of code**
- **hides underlying implementation of inverted lists: disk layout and index compression**
- **allows optimizations similar to zig-zag join in DBs** ***(can use a seek forward)***
- **should be implemented by you in homework #2, using BM25**

Inverted Index Compression



- idea: use efficient coding for docIDs, frequencies, and positions in index
- first, take differences, then encode those smaller numbers:
- example: encode alligator list, first produce differences:

  - if postings only contain docID:

    (34) (68) (131) (241) … becomes (34) (34) (43) (110) …
  - if postings with docID and frequency:

    (34,1) (68,3) (131,1) (241,2) … becomes (34,1) (34,3) (43,1) (110,2) …
  - if postings with docID, frequency, and positions:

    (34,1,29) (68,3,9,46,98) (131,1,46) (241,2,45,131) …

    becomes (34,1,29) (34,3,9,37,52) (43,1,46) (110,2,45,86) …
  - afterwards, do encoding with one of many possible methods

Simple Method: vbyte

- simple byte-oriented method for encoding data
- encode number as follows:
  - if < 128, use one byte (highest bit set to 0)
  - if < 128*128 = 16384, use two bytes (first has highest bit 1, the other 0)
  - if < 128^3, then use three bytes, and so on …
- examples: 14169 = 110*128 + 89 = $\boxed{11101110}\boxed{01011001}$

  33549 = 2*128*128 + 6*128 + 13 = $\boxed{10000010}\boxed{10000110}\boxed{00001101}$
- example for a list of 4 docIDs: after taking differences

  (34) (178) (291) (453) … becomes (34) (144) (113) (162)
- this is then encoded using six bytes total:

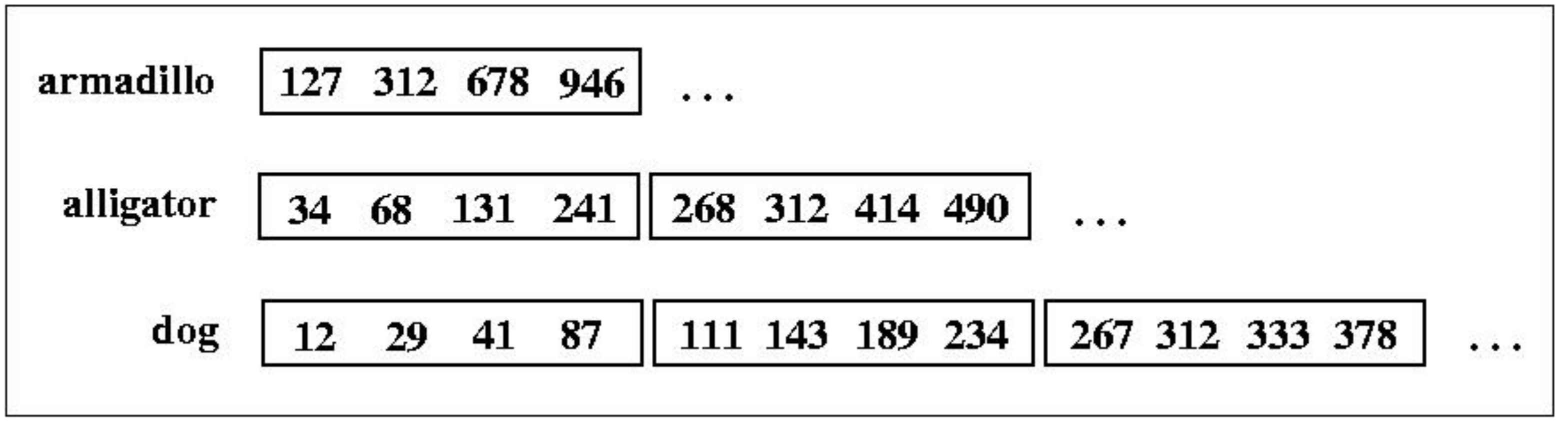
  34 = $\boxed{00100010}$

  144 = $\boxed{10000001}\boxed{00010000}$

  113 = $\boxed{01110001}$

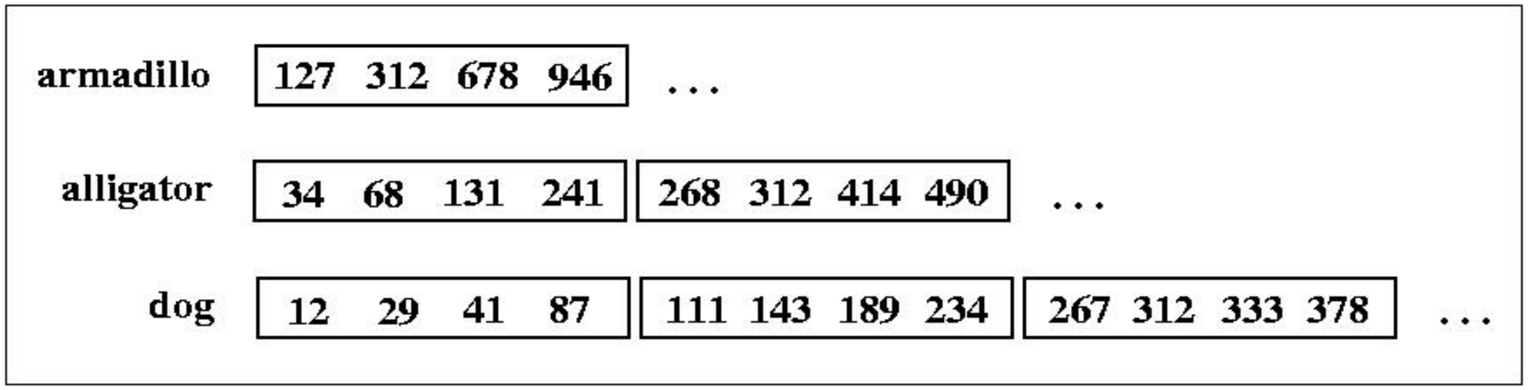
  162 = $\boxed{10000001}\boxed{00100010}$
- not a great encoding, but fast and reasonably OK
- implement using char array and char* pointers in C/C++

Chunkwise Compression



- in real systems, compression is done in chunks
- each chunk can be individually decompressed
- this allows nextGEQ to jump forward without uncompressing all entries, by skipping over entire blocks
- this requires an extra auxiliary table, say containing the docID of the last posting in each chunk and the size of each chunk
- chunks may be fixed size of fixed number of postings (e.g, each chunk 256 bytes, or each chunk 128 postings)
- details/choices depend on various issues:
  (compression technique used, posting format, cache line alignment, wasted space)

Implementing nextGEQ()



- **no chunkwise compression:**
  - simple loop
  - uncompress all until found

```
int nextGEQ(lp, k)
{ while (lp->did < k)
     lp->did = decodeNext(lp);
  return(lp->did);      }
```

- **with chunkwise compression:**
  - extra array last[]
  - contains last docID in each chunk
  - for dog, contains 87, 234, 378
  - uncompress() checks if block is already uncompressed, else it uncompresses docIDs into temp[] and sets lp->ind to zero

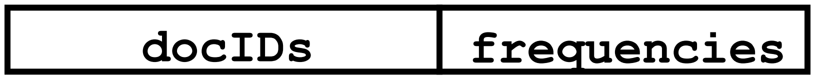
```
int nextGEQ(lp, k)
{ while (last[lp->block] < k)
     lp->block++;   /* skip block */

  uncompress(lp->block, temp);
  while (temp[lp->ind] < k)
     lp->ind++;
  return(temp[lp->ind]); }
```

- **warning: slightly simplified pseudocode!**

More Details: (blockwise compression)

- in general, the lp listpointer struct maintains various state
  - information about the list and term itself (e.g., global term frequency)
  - pointers to the “current” posting in the list and/or its docID
  - info about whether the current chunk has already been uncompressed
- note: decompression of chunks often more efficient than decompression of postings one by one (using decodeNext())
- in each chunk, may separate docIDs from freqs and posits:

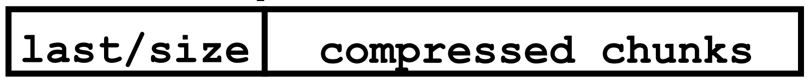
e.g., chunk of 128 postings:



- only decompress frequencies if docID in intersection found
- even if we decompress docIDs of block, we often can skip decompression of frequencies
- blockwise compression allows skipping of most blocks in the longer inverted lists during a query

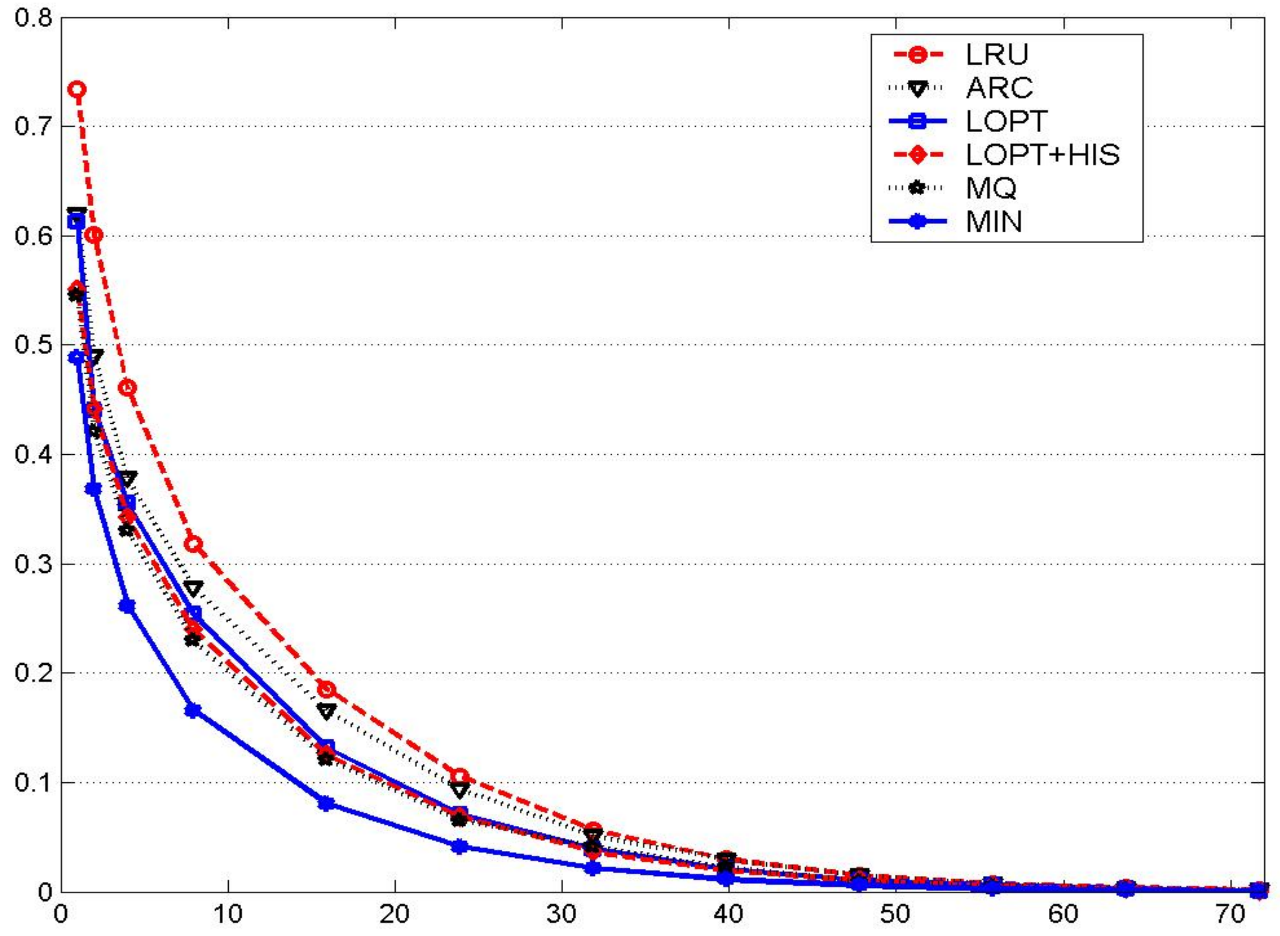
Inverted List Caching:

- inverted lists are on disk, and fetched during processing of query
- lists long, so disk can become major bottleneck
- idea: cache inverted lists for common query words in memory!
- implementation: partition index into blocks, say 64KB/block, and treat each block separately as an object during caching
- alternative: treat lists as objects (but this would have some problems)
- thus, each block contains say a few hundred chunks, from one list or from several different lists
- put last[] and size[] data for blockwise decompression into first part of each block



- always prefetch next few blocks (say, up to 2MB of data)
- all blocks in list are fetched from disk unless already cached (forward seeks due to nextGEQ() are rarely long enough to skip entire blocks)

Caching of Inverted Lists in Memory



- based on query trace of 1.8 million Excite queries (1999 and 2001)
- e.g., MQ method gets around 12% cache miss rate (88% hit rate) if only 16% of the inverted index fits into the cache (see 4th data point from left)
- LRU not the best method (MIN is "optimal" method: infeasible, clairvoyant)
- caching makes a huge difference in performance!

```
for (i = 0; i < numterms; i++)  lp[i] = openList(qterm[i]);

for (docid = 0; docid < MAXDOCID; docid++)
{
  /* get next element from first (shortest) list */
  docid = nextGEQ(lp[0], docid);

  /* see if you find entries with same docID in other lists */
  for (i = 1, d = docid; (i < numterms) && (d == docid); i++)
    d = nextGEQ(lp[i], docid);

  if (d > docid)    /* docid not in intersection; continue */
    docid = d-1;
  else              /* docid in intersection; compute score */
  {
    for (i = 0; i < numterm; i++)  p[i] = getPost(lp[i], did);
    computeScore(p, numterm);
  }
}
for (i = 0; i < num; i++)  closeList(lp[i]);
```