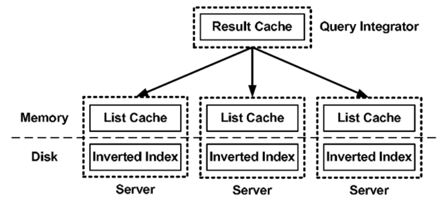
Recap: Search Engine Query Processing

- Basically, to process a query we need to traverse the inverted lists of the query terms
- Lists are very long and are stored on disks
- Challenge: traverse lists as quickly as possible
- Tricks: compression, caching,
  parallelism, early termination ("pruning")

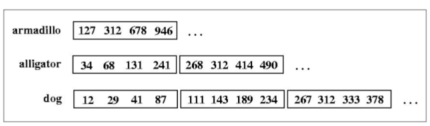
polytechnic	127 312 678 946 …
university	34 168 188 312 467 787 946 …
brooklyn	25 38 95 127 178 188 203 296 …

Recap: Search Engine Query Processing

- Parallel query processing: divide docs between many machines, broadcast results to all
- Caching of results at query integrator
- Caching of compressed lists at each node

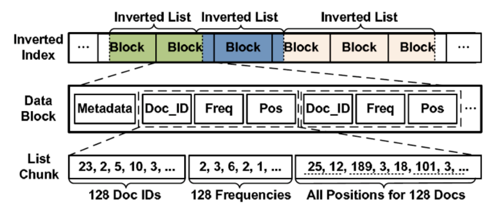


Chunked Compression



- 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 containing the docID of the last posting in each chunk (and maybe another one with the size of each chunk)
- Chunks may be fixed size or fixed number of postings (e.g, each chunk 256 bytes, or each chunk 128 postings)
  Issues: compression technique, posting format, cache line alignment, wasted space

Index Structure Layout



- Data blocks, say of size 64KB, as basic unit for list caching
- List chunks, say of 128 postings, as basic unit of decompression
- Many chunks are skipped over, but very few blocks are
- Also, may prefetch the next, say 2MB of index data from disk

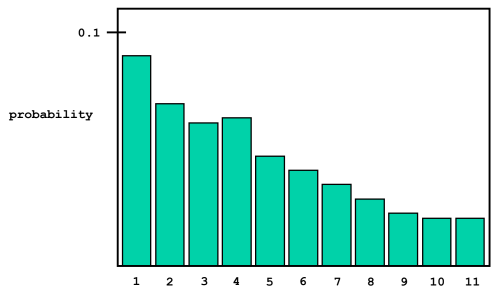
Inverted List Compression Techniques

- Inverted lists:
  - consist of docIDs, frequencies, positions (also context?)
  - basically, integer values
  - most lists are short, but large lists dominate index size
- How to compress inverted lists:
  - for docIDs, positions: first "compute differences" (gaps)
  - this makes docIDs, positions smaller (freqs already small)
  - problem: "compressing numbers that tend to be small"
  - need to model the gaps, i.e., exploit their characteristics
- And remember: usually done in chunks
- Local vs. global methods
- Exploiting clustering of words: book vs. random page order

Techniques Covered in this Class

- Simple and OK, but not great:
  - vbyte (var-byte): uses variably number of bytes per integer
- Better compression, but slower than var-byte:
  - Rice Coding and Golomb Coding: bit oriented
  - use statistics about average or median of numbers (gap size)
- Good compression for very small numbers, but slow:
  - Gamma Coding and Delta Coding: bit oriented
  - or just use Huffman?
- Better compression than VByte, and *REALLY* fast:
  - Simple9 (Anh/Moffat 2001): pack as many numbers as possible in 32 bits (one word)
  - PFOR-DELTA (Heman 2005): compress, e.g., 128 number at a time. Each number either fixed size, or an exception.

Distribution of Integer Values



- many small values means better compression

Recap: Taking Differences

armadillo [127 312 678 946] ...

alligator [34 68 131 241] [268 312 414 490] ...

dog [12 29 41 87] [111 143 189 234] [267 312 333 378] ...

- 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

Recap: var-byte Compression

- 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 = [11101110][01011001]
  33549 = 2*128*128 + 6*128 + 13 = [10000010][10000110][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 = [00100010]
  144 = [10000001][00010000]
  113 = [01110001]
  162 = [10000001][00100010]
- not a great encoding, but fast and reasonably OK
- implement using char array and char* pointers in C/C++

Rice Coding:

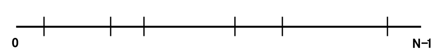
- consider the average or median of the numbers (i.e., the *gaps*)
- simplified example for a list of 4 docIDs: after taking differences
  (34) (178) (291) (453) ... becomes (34) (144) (113) (162)
- so average is g = (34+144+113+162) / 4 = 113.33
- Rice coding: round this to smaller power of two: b = 64 (6 bits)
- then for each number x, encode x-1 as
  (x-1)/b in unary followed by (x-1) mod b binary (6 bits)
  33 = 0*64+33 = 0 100001
  143 = 2*64+15 = 110 001111
  112 = 1*64+48 = 10 110000
  161 = 2*64+33 = 110 100001
- note: there are no zeros to encode (might as well deduct 1 everywhere)
- simple to implement (bitwise operations)
- better compression than var-byte, but slightly slower

Golomb Coding:

- example for a list of 4 docIDs: after taking differences
  (34) (178) (291) (453) ... becomes (34) (144) (113) (162)
- so average is g = (34+144+113+162) / 4 = 113.33
- Golomb coding: choose b ~ 0.69*g = 78 (usually not a power of 2)
- then for each number x, encode x-1 as
  (x-1)/b in unary followed by (x-1) mod b in binary (6 or 7 bits)
- need fixed encoding of number 0 to 77 using 6 or 7 bits
- if (x-1) mod b < 50: use 6 bits else: use 7 bits
- e.g., 50 = 110010 0 and 64 = 110010 1
  33 = 0*78+33 = 0 100001
  143 = 1*78+65 = 10 1100111
  112 = 1*78+34 = 10 100010
  161 = 2*78+5 = 110 000101
- optimal for random gaps (dart board, random page ordering)

Rice and Golomb Coding:

- uses parameters b – either global or local
- local (once for each inverted list) vs. global (entire index)
- local more appropriate for large index structures
- but does not exploit clustering within a list
- compare: random docIDs vs. alpha-sorted vs. pages in book
  - random docIDs: no structure in gaps, global is as good as local
  - pages in book: local better since some words only in certain chapters
  - assigning docIDs alphabetically by URL is more like case of a book
- instead of storing b, we could use N (# of docs) and f_t :
  $g = (N - f_t) / (f_t + 1)$
- idea: e.g., 6 docIDs divide 0 to N-1 into 7 intervals



Gamma and Delta Coding:

- no parameters such as b: each number coded by itself
- simplified example for a list of 4 docIDs: after taking differences
  (34) (178) (291) (453) … becomes (34) (144) (113) (162)
- imagine each number as binary with leading 1: 34 = 100010
- then for each number x, encode x-1 as
  1 + floor(log(x)) in unary followed by floor(log(x)) bits
- thus, 1 = 0 and 5 = 110 01
  33 = 111110 00001
  143 = 11111110 0001111
  112 = 1111110 110000
  161 = 11111110 0100001
- note: good compression for small values, e.g., frequencies
- bad for large numbers, and fairly slow
- Delta coding: Gamma code; then gamma the unary part

Simple9 (S9) Coding: (Anh/Moffat 2004)

- idea: produce a word-aligned code – basic unit 32 bits
- try to pack several numbers into one word (32 bits)
- each word is split into 4 control bits and 28 data bits
- what can we store in 28 bits?
  - 1 28-bit number
  - 2 14-bit numbers
  - 3 9-bit numbers (1 bit wasted)
  - 4 7-bit numbers
  - 5 5-bit numbers (3 bits wasted)
  - 7 4-bit numbers
  - 9 3-bit numbers (1 bit wasted)
  - 14 2-bit numbers
  - 28 1-bit numbers
- then use other 4 bits to store which of these 9 cases is used
  (assumption for simplicity: all numbers that we encounter need at most 28 bits)

Simple9 (S9) Coding: (continued)

- store and retrieve numbers using fixed bit masks
- algorithm:
  - do the next 28 numbers fit into one bit each?
    - if yes: use that case
    - if no: do the next 14 numbers fit into 2 bits each?
      - if yes: use that case
      - if no: do the next 9 numbers fit into 3 bits each?
        … and so on …
- fast decoding: only one if-decision for every 32 bits
- compare to varbyte: one or more decisions per number
- decent compression: can use < 1 byte for small numbers
- related techniques: relate10 and carryover12
- Simple16 (S16): contains several optimizations over S9

PFOR-DELTA: (Heman 2005)

- idea: compress/decompress many values at a time (e.g., 128)
- how many bits per number?
  - different choice for each number? (decoding slow due to branches)
  - or one size fits all? (bad compression)
- good compromise: choose size such that 90% fit, code the other 10% as exceptions
- suppose in next 128 numbers, 90% are < 32 : choose b=5
- allocate 128 x 5 bits, plus space for exceptions
- exceptions stored at end as ints (using 4 bytes each)
- example: b=5 and sequence 23, 41, 8, 12, 30, 68, 18, 45, 21, 9, ..

$\boxed{1}$	23	3	8	12	30	1	18	2	21	9				…	45	68	41

stores location of 1st exception — space for 128 5-bit numbers — space for exceptions (4 bytes each, back to front)

- exceptions (grey) form linked list within the locations (e.g., 3 means "next except. 3 away")
- one extra slot at beginning points to location of first exception (or store in separate array)

PFOR-DELTA: (ctd.)

- there may sometimes be "forced exceptions":
  in example: if there are more than 2^b consecutive numbers < 2^b, then encode the 2^b-th number as exception so we can keep a simple linked list structure
- very simple and fast decoding
  - first, copy the 128 b-bit numbers into integer array (very fast per element)
  - then traverse linked list and patch the exceptions (slower per element)
  - if we keep exceptions < 10%, this will be extremely fast
  - first phase: unroll loops for best performance – hardcode for each b
- note: always uncompress next 128 posts into temp array
  - do not uncompress entire list into one long array: slower since out of cache
- simple effective improvement: do not use 32 bits / except
  - use maximum among next 128 numbers to choose number of bits
  - 10-20% better compression with basically same speed (if done properly)

$\boxed{1}$	23	3	8	12	30	1	18	2	21	9				…	45	68	41

stores location of 1st exception — space for 128 5-bit numbers — space for exceptions (32 bits each)

Some Experimental Numbers

- results from Witten/Moffat/Bell book
- includes golomb, gamma, delta, but not others above
- data with "locality": books, or web pages sorted by URL
  - word occurrences not uniform within a book, but often clustered in one part
- in this case, interpolative better
- see book for details

FROM: WITTEN/MOFFAT/BELL : MANAGING GBs

Table 3.1 Statistics of document collections.

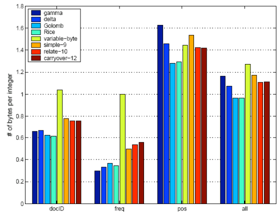
		Collection			
		Bible	*GNUbib*	*Comact*	*TREC*
Documents	N	31,101	64,343	261,829	741,856
Number of terms	F	884,994	2,570,906	22,805,920	333,338,738
Distinct terms	n	8,965	46,488	36,660	535,346
Index pointers	f	701,412	2,226,300	12,976,418	134,994,414
Total size (Mbytes)		4.33	14.05	131.86	2070.29

Table 3.8 Compression of inverted files in bits per pointer.

Method	Bits per pointer			
	Bible	*GNUbib*	*Comact*	*TREC*
Global methods				
Unary	262	909	487	1918
Binary	15.00	16.00	18.00	20.00
Bernoulli	9.86	11.06	10.90	12.30
γ	6.51	5.68	4.48	6.63
δ	6.23	5.08	4.35	6.38
Observed frequency	5.90	4.82	4.20	5.97
Local methods				
Bernoulli	6.09	6.16	5.40	5.84
Hyperbolic	5.75	5.16	4.65	5.89
Skewed Bernoulli	5.65	4.70	4.20	5.44
Batched frequency	5.58	4.64	4.02	5.41
Interpolative	5.24	3.98	3.87	5.18

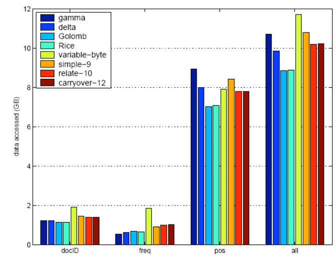
Some Newer Experimental Numbers

- by Xiaohui Long, 2006
- includes golomb, rice, gamma, delta, S9 and its variants
- lists weighted by frequency in queries
  - not total index size, but size of compressed data fetched per query
  - but also tracks index size reasonably well
- bytes per compressed integer in list
- var-byte bad for frequency
  - always at least one byte
- S9 and variants much better
- but not as good as others



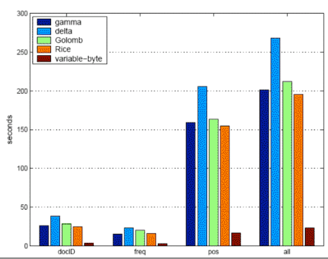
Some Experimental Numbers (ctd.)

- another perspective: index data access in GB / 1000 queries
- note: position data much larger than docID and frequency
  reason: several positions/posting, and larger numbers on average
- relative differences in cost smaller if we have positions



Some Experimental Numbers (ctd.)

- CPU cost for uncompression (Xiaohui Long, 2006)
- cost per 1000 queries on 8 million pages (not fully optimized)
- var-byte MUCH faster than the others
- later: other newer techniques (S9, PFORDELTA, etc.) also fast



Hacking up Rice Coding:

- can we implement Rice coding much faster than known?
- note similarity to PFORDELTA: unary part == exception
- more bits for binary part == fewer exceptions
- idea: when compressing 128 integers:
  - store 128 binary parts followed by 128 unary parts
  - during decompression, first retrieve the 128 binary parts
  - use same bit-copy routines as in PFORDELTA
  - then apply unary parts to patch things up
  - of course, more exceptions as in PFORDELTA
- second idea: process 8 bits of the unary data at once
  - switch statement with 256 cases and 2000 lines of code - but fast!

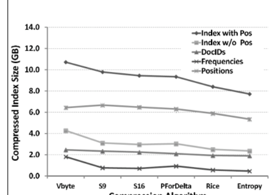
Experimental Setup:

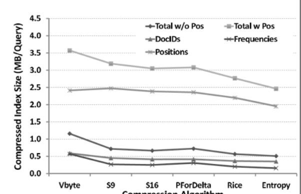
- set of 7.4 million web pages
- Excite query trace from 1999

Trace	Queries	Unique Queries	Query Length	List Length
Excite	1,500,005	536,239	2.59	220,331
AOL(time)	1,861,054	536,239	2.75	208,426
AOL(user)	1,920,154	536,239	2.80	204,663

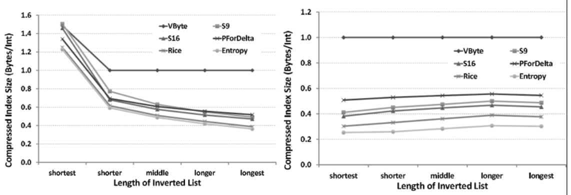
- remove duplicate queries (to take result caching into account)
- select 1000 consecutive queries, run in main memory
- 3.2 Ghz Pentium 4, gcc compiler, …
- used var-byte for very short lists

Compressed Size:





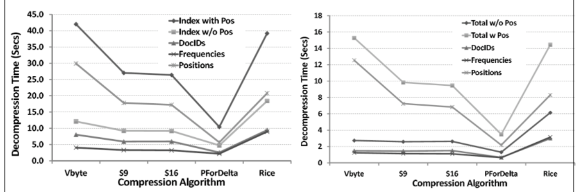
Bytes per Integer:



docIDs

frequencies

Decompression Times:



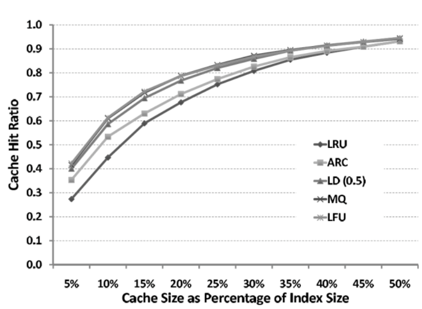
Decompression Speeds: (millions of integers / second)

Algorithm	Total w/o Pos	Total w Pos	DocID	Freq	Pos
VByte	416.32	183.66	381.90	457.56	132.86
S9	439.49	285.27	391.73	500.51	230.04
S16	433.78	296.53	376.92	510.86	243.76
PForDelta	868.70	803.11	855.79	882.01	763.67
Rice	185.39	194.20	190.44	180.59	200.72

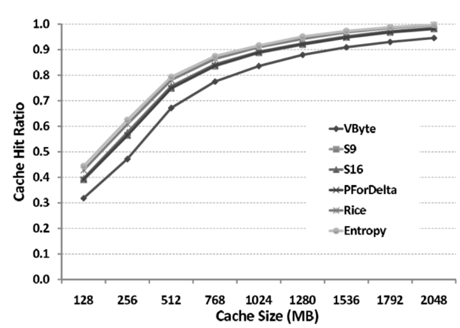
Index Caching - Algorithms

- study of replacement policies for list caching
- most common algorithm: LRU (Least Recently Used)
- alternative: LFU (Least Frequently Used)
- discussion: LRU vs. LFU
  - LRU good for changing hot items, LFU for more static
  - out of cache, out of mind ?
- Landlord: generalization of weighted caching
  - analyzed for weighted caching (Cao/Irani/Young)
  - modification: give longer leases to repeat tenants
- Multi-Queue (MQ) (Zhou/Philbin/Li 2001)
- Adaptive Replacement Policy (Megiddo/Modha 2003)

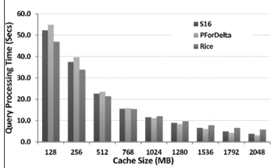
Comparison of Caching Policies:



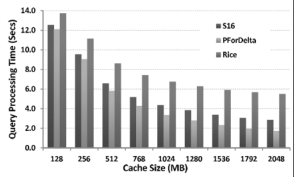
Impact of Compression:



Total Cost for Fixed Disk Speed:

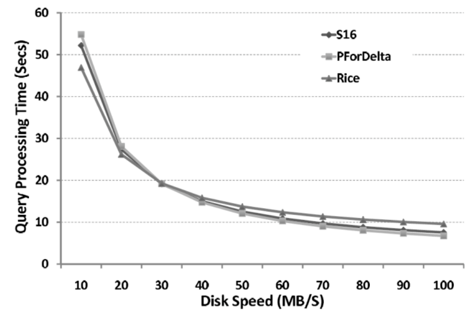


10 MB/s disk



50 MB/s disk

Effect of Disk Speed:



Conclusions

- Great differences in speed and compression
- Old story: var-byte is not as good in compression, but much faster and thus used in practice
- New story (last 2-3 years): there are other techniques that are faster and also compress much better
- Decompression speeds: GBs per second !
- Bit- versus byte-alignment is not the issue
- But you need to be able to use fixed masks and avoid branch mispredicts (simple ideas, long code)
- LRU not a good caching policy
- Compression has caching consequences …
- Better compression gives higher cache hit ratio

Index Compression in Google (1998)

- see paper for details
- forward barrel: postings during sorting, before final index constructed
- inverted barrels: inverted index structure: 27 bits / docID, 5 bits / freq
- plus extra context data about each hit (each occurrence)
- was replaced by newer technique …

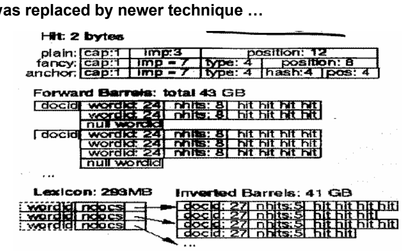


Figure 3. Forward and Reverse Indexes and the Lexicon