Web Information Retrieval

Textbook by
Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schutze
Notes Revised by X. Meng for SEU
May 2014

Recap

- Index construction in memory
- Boolean query and retrieval

Creating an Inverted Index

Create an empty index term list I;
For each document, D, in the document set V
    For each (non-zero) token, T, in D:
        If T is not already in I
            Insert T into I;
        Find the location for T in I;
        If (T, D) is in the posting list for T
            increase its term frequency for T;
        Else
            Create (T, D);
            Add it to the posting list for T;

Optimized Intersection Algorithm for Conjunctive Queries

```
INTERSECT(⟨t1, ..., tn⟩)
1  terms ← SORTBYINCREASINGFREQUENCY(⟨t1, ..., tn⟩)
2  result ← postings(first(terms))
3  terms ← rest(terms)
4  while terms ≠ NIL and result ≠ NIL
5  do result ← INTERSECT(result, postings(first(terms)))
6     terms ← rest(terms)
7  return result
```


4

Outlines

- We will discuss two topics.
  - Index construction
  - Index compression

Index Construction Review

- We've discussed the basic algorithm for index construction:

Create an empty index term list I;
For each document, D, in the document set V
    For each (non-zero) token, T, in D:
        If T is not already in I
            Insert T into I;
        Find the location for T in I;
        If (T, D) is in the posting list for T
            increase its term frequency for T;
        Else
            Create (T, D);
            Add it to the posting list for T;

Discuss the Simple Indexing Algorithm

- Everything in SIA is in memory while in reality the data indexed by a commercial search engine is much bigger that will not fit in memory
  - Solution: read and write information from and to secondary storage (e.g., disks)
- Even with secondary storage, we still need to consider the issue of index compression

Hardware Basics

- Many design decisions in information retrieval are based on hardware constraints.
- We begin by reviewing hardware basics that we'll need in this course.

Hardware Basics

- Access to data is much faster in memory than on disk. (roughly a factor of 1000)
- Disk seeks are "idle" time: No data is transferred from disk while the disk head is being positioned.
- To optimize transfer time from disk to memory: one large chunk is faster than many small chunks.

Hardware Basics

- Disk I/O is block-based: Reading and writing of entire blocks (as opposed to smaller chunks). Block sizes: 8KB to 256 KB (Google file system blocks are 64 MB in size)
- Servers used in IR systems typically have several GB of main memory, sometimes tens of GB, and TBs or 100s of TB of disk space.
- Fault tolerance is expensive: It's cheaper to use many regular machines than one fault tolerant machine.

Some Stats (ca. 2008)

symbol	statistic	value
s	average seek time	5 ms = 5×10^{-3} s
b	transfer time per byte	0.02 µs = 2×10^{-8} s
	processor's clock rate	10^9 s^{-1}
P	Low level operation (e.g., compare & swap a word)	0.01 µs = 10^{-8} s
	size of main memory	several GB
	size of disk space	1 TB or more

Some Definitions Used in the Textbook

- *Token*: a useful unit for processing, such as "word," "2013," or "LLC"
- *Type*: a class of all tokens containing the same character sequence
- *Term*: a type that is included in the IR system
- For example, "to be, or not to be: that is the question."
  - 10 tokens
  - 8 types (to, be, or, not, that, is, the, question)
  - 1 term, if "to, be, or, not, that, is, the" are considered stopwords

RCV1 Collection

- RCV: Reuters (路透社) Corpus Volume, English news article collections between 1995 and 1996
- Shakespeare's collected works are not large enough for demonstrating many of the points in this course.
- As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.

13 13

A Reuters RCV1 Document


REUTERS

You are here: Home > News > Science > Article

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly En

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprir

[-] Text [-

SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian

14 14

Reuters RCV1 statistics

N	documents	800,000
L	tokens per document	200
M	terms (= word types)	400,000
	bytes per token (incl. spaces/punct.)	6
	bytes per token (without spaces/punct.)	4.5
	bytes per term (= word type)	7.5
T	non-positional postings	100,000,000

Average frequency of a term (how many tokens)? 4.5 bytes per word token vs. 7.5 bytes per word type: why the difference?

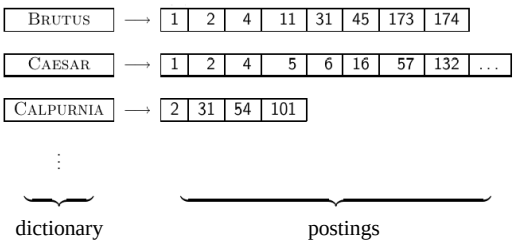
15 15

Outline

- Recap
- Introduction
- Blocked sort-based indexing (BSBI) algorithm
- SPIMI algorithm
- Distributed indexing
- Dynamic indexing

16

Goal: Construct the Inverted Index



17 17

Index Construction in IIR 1: Sort Postings in Memory

term	docID
i	1
did	1
enact	1
julius	1
caesar	1
i	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

⟹

term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
i	1
i	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

18 18

Sort-based Index Construction

- As we build index, we parse docs one at a time.
- The final postings for any term are incomplete until the end of the entire document collection.
- Can we keep all postings in memory and then do the sort in-memory at the end?
- No, not for large collections.

Why?

- At 10–12 bytes per postings entry, we need a lot of space for large collections.
- T = 100,000,000 in the case of RCV1: we can do this in memory on a typical machine in 2010.
- But in-memory index construction does not scale for large collections.
- Thus: We need to store intermediate results on disk.

Same Algorithm for Disk?

- Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?
- No: Sorting T = 100,000,000 records on disk is too slow – too many disk seeks.
- We need an external sorting algorithm.

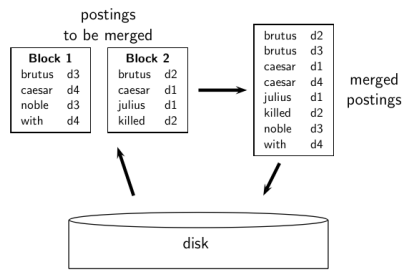
"External" Sorting Algorithm (Using Few Disk Seeks)

- We must sort T = 100,000,000 non-positional postings.
  - Each posting has size 12 bytes (4+4+4: termID, docID, document frequency).
- Define a block to consist of 10,000,000 such postings
  - Assume we can fit that many postings into memory.
  - We will have 10 such blocks for RCV1.

"External" Sorting Algorithm (Using Few Disk Seeks)

- Basic idea of algorithm:
  - For each block:
    - (i) accumulate postings,
    - (ii) sort in memory,
    - (iii) write to disk
  - Then merge the blocks into one long sorted order.

Merging Two Blocks



Blocked Sort-Based Indexing (BSBI)

```
BSBIndexConstruction()
1  n ← 0
2  while (all documents have not been processed)
3  do n ← n + 1
4     block ← ParseNextBlock()
5     BSBI-Invert(block)
6     WriteBlockToDisk(block, f_n)
7  MergeBlocks(f_1, ..., f_n; f_merged)
```

– Key decision: What is the size of one block?

25 25

Outline

- Recap
- Introduction
- BSBI algorithm
- Single Pass In-Memory Index (SPIMI) algorithm
- Distributed indexing
- Dynamic indexing

26

Problem with Sort-Based Algorithm

- Our assumption was: we can keep the dictionary in memory.
- We need the dictionary (which grows dynamically) in order to implement a term to termID mapping.
- Actually, we could work with term, docID postings instead of termID, docID postings . . .
- . . . but then intermediate files become very large. (We would end up with a scalable, but very slow index construction method.)

27 27

Single-Pass In-Memory Indexing

- Abbreviation: SPIMI
- Key idea 1: Generate separate dictionaries for each block – no need to maintain term-termID mapping across blocks.
- Key idea 2: Don't sort. Accumulate postings in postings lists as they occur.
- With these two ideas we can generate a complete inverted index for each block.
- These separate indexes can then be merged into one big index.

28 28

SPIMI-Invert Algorithm

```
SPIMI-Invert(token_stream)
1  output_file ← NewFile()
2  dictionary ← NewHash()
3  while (free memory available)
4  do token ← next(token_stream)
5     if term(token) ∉ dictionary
6        then postings_list ← AddToDictionary(dictionary, term(token))
7        else postings_list ← GetPostingsList(dictionary, term(token))
8     if full(postings_list)
9        then postings_list ← DoublePostingsList(dictionary, term(token))
10    AddToPostingsList(postings_list, docID(token))
11 sorted_terms ← SortTerms(dictionary)
12 WriteBlockToDisk(sorted_terms, dictionary, output_file)
13 return output_file
```

Merging of blocks is analogous to BSBI.

29 29

SPIMI: Compression

- Compression makes SPIMI even more efficient.
  - Compression of terms
  - Compression of postings
  - See next lecture

30 30

Outline

- Recap
- Introduction
- BSBI algorithm
- SPIMI algorithm
- Distributed indexing
- Dynamic indexing

31

Distributed Indexing

- For web-scale indexing (don't try this at home!): must use a distributed computer cluster
- Individual machines are fault-prone.
  - Can unpredictably slow down or fail.
- How do we exploit such a pool of machines?

32 32

Search Engine Data Centers

- In order to handle such huge amount of data, search engine companies establish a number of data centers across the globe, see a separate lecture on the subject of data centers

33 33

Distributed Indexing

- Maintain a master machine directing the indexing job – considered "safe"
- Break up indexing into sets of parallel tasks
- Master machine assigns each task to an idle machine from a pool.

34 34

Parallel Tasks

- We will define two sets of parallel tasks and deploy two types of machines to solve them:
  - Parsers
  - Inverters
- Break the input document collection into splits (corresponding to blocks in BSBI/SPIMI)
- Each split is a subset of documents.

35 35

Parsers

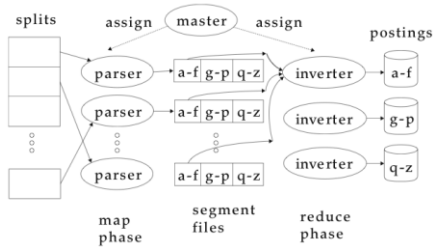
- Master assigns a split to an idle parser machine.
- Parser reads a document at a time and emits (term,docID)-pairs.
- Parser writes pairs into j term-partitions.
- Each for a range of terms' first letters
  - E.g., a-f, g-p, q-z (here: j = 3)

36 36

Inverters

- An inverter collects all (term,docID) pairs (= postings) for one term-partition (e.g., for a-f).
- Sorts and writes to postings lists

Data Flow



MapReduce

- The index construction algorithm we just described is an instance of MapReduce.
- MapReduce is a robust and conceptually simple framework for distributed computing . . .
- . . .without having to write code for the distribution part.
- Just like sorting, search algorithms, MapReduce is becoming a critical computation model for distributed computing

A Simple Map-Reduce Example

- Sum up a sequence of n numbers on k processors
- The Map phase:
  - Divide the n numbers into k sets
  - Each processor adds up n/k numbers and store the partial sum as s_k
- The Reduce phase:
  - Add up the partial sums s_k to get the total sum s

MapReduce

- The Google indexing system (ca. 2002) consisted of a number of phases, each implemented in MapReduce.
- Index construction was just the first phase.
- The second phase: transform term-partitioned into document-partitioned index.

Index Construction in MapReduce

Schema of map and reduce functions

map: input → list(k, v)
reduce: (k,list(v)) → output

Instantiation of the schema for index construction

map: web collection → list(termID, docID)
reduce: (⟨termID$_1$, list(docID)⟩, ⟨termID$_2$, list(docID)⟩, . . .) → (postings_list$_1$, postings_list$_2$, . . .)

Example for index construction

map: d_2 : C DIED. d_1 : C CAME, C C'ED. → (⟨C, d_2⟩, ⟨DIED,d_2⟩, ⟨C,d_1⟩, ⟨CAME,d_1⟩, ⟨C,d_1⟩, ⟨C'ED,d_1⟩)
reduce: (⟨C,⟨d_2,d_1,d_1⟩⟩,⟨DIED,⟨d_2⟩⟩,⟨CAME,⟨d_1⟩⟩,⟨C'ED,⟨d_1⟩⟩) → (⟨C,⟨d_1:2,d_2:1⟩⟩,⟨DIED,⟨d_2:1⟩⟩,⟨CAME,⟨d_1:1⟩⟩,⟨C'ED,⟨d_1:1⟩⟩)

Outline

- Recap
- Introduction
- BSBI algorithm
- SPIMI algorithm
- Distributed indexing
- Dynamic indexing

Dynamic Indexing

- Up to now, we have assumed that collections are static.
- They rarely are: Documents are inserted, deleted and modified.
- This means that the dictionary and postings lists have to be dynamically modified.

44

Dynamic Indexing: Simplest Approach

- Maintain big main index on disk
- New docs go into small auxiliary index in memory.
- Search across both, merge results
- Periodically, merge auxiliary index into big index
- Deletions:
  - Invalidation bit-vector for deleted docs
  - Filter docs returned by index using this bit-vector

45

Issue with Auxiliary and Main Index

- Frequent merges
- Poor search performance during index merge
- Actually:
  - Merging of the auxiliary index into the main index is not that costly if we keep a separate file for each postings list.
  - Merge is the same as a simple append.
  - But then we would need a lot of files – inefficient.

46

Issue with Auxiliary and Main Index

- Assumption for the rest of the lecture: The index is one big file.
- In reality: Use a scheme somewhere in between (e.g., split very large postings lists into several files, collect small postings lists in one file etc.)
- Time complexity:
  - index construction time is $O(T^2)$ as each posting is touched in each merge.
  - Suppose auxiliary index has size a
  - $a + 2a + 3a + 4a + \ldots + na = a\frac{n(n+1)}{2} = O(n^2)$

47

Logarithmic Merge

- Logarithmic merging amortizes the cost of merging indexes over time.
  - Users see smaller effect on response times.
- Maintain a series of indexes, each twice as large as the previous one.
- Keep smallest (Z0) in memory
- Larger ones (I0, I1, . . .) on disk
- If Z0 gets too big (> n), write to disk as I0
- . . . or merge with I0 (if I0 already exists) and write merger to I1 etc.

48

The Algorithm

```
LMERGEADDTOKEN(indexes, Z0, token)
 1  Z0 ← MERGE(Z0, {token})
 2  if |Z0| = n
 3    then for i ← 0 to ∞
 4         do if Ii ∈ indexes
 5               then Zi+1 ← MERGE(Ii, Zi)
 6                    (Zi+1 is a temporary index on disk.)
 7                    indexes ← indexes − {Ii}
 8               else Ii ← Zi    (Zi becomes the permanent index Ii.)
 9                    indexes ← indexes ∪ {Ii}
10                    BREAK
11         Z0 ← ∅

LOGARITHMICMERGE()
1  Z0 ← ∅    (Z0 is the in-memory index.)
2  indexes ← ∅
3  while true
4  do LMERGEADDTOKEN(indexes, Z0, GETNEXTTOKEN())
```

49

Logarithmic Merge

- Number of indexes bounded by O(log T) (T is total number of postings read so far)
- So query processing requires the merging of O(log T) indexes
- Time complexity of index construction is O(T log T).
- . . . because each of T postings is merged O(log T) times.
- So logarithmic merging is an order of magnitude more efficient than the simple merge seen earlier.
- The price we pay is a bit slower in search operations

50 50

Dynamic Indexing at Large Search Engines

- Often a combination of
  - Frequent incremental changes
  - Rotation of large parts of the index that can then be swapped in
  - Occasional complete rebuild (becomes harder with increasing size – not clear if Google can do a complete rebuild)

51 51

Building Positional Indexes

- Basically the same problem except that the intermediate data structures are much larger.

52 52

Compression

Index compression and posting lists compression

53

Compression

- Inverted lists are very large
  - e.g., index lists occupy 25-50% of the collection for TREC collections using Indri search engine
  - Much higher if n-grams are indexed
- Compression of indexes saves disk and/or memory space
  - Typically have to decompress lists to use them
  - Best compression techniques have good *compression ratios* and are easy to decompress
- *Lossless* compression – no information lost

Compression

- *Basic idea*: Common data elements use short codes while uncommon data elements use longer codes
  - Example: coding numbers
    - number sequence: $0, 1, 0, 3, 0, 2, 0$
    - possible encoding (14 bits): 00 01 00 10 00 11 00
    - encode 0 using a single 0: 0 01 0 10 0 11 0
    - only 10 (ten) bits, but...

Compression Example

- *Ambiguous* encoding – not clear how to decode
  - the same coding: 0 01 01 0 0 11 0
  - which represents: $0, 1, 1, 0, 0, 3, 0$
  - use unambiguous code:
    - example: *prefix codes*
  - which gives: (13 bits)

Number	Code
0	0
1	101
2	110
3	111

0 101 0 111 0 110 0

Recap

- Introduction
- BSBI algorithm
- SPIMI algorithm
- Distributed indexing (MapReduce)
- Dynamic indexing

57

Delta Encoding

- Word count data is good candidate for compression
  - many small numbers and few larger numbers
  - encode small numbers with small codes
- Document numbers (docID) are less predictable
  - but differences between numbers in an ordered list are smaller and more predictable
- *Delta encoding*:
  - encoding differences between document numbers (*d-gaps*)

Delta Encoding

- Inverted list

  $1, 5, 9, 18, 23, 24, 30, 44, 45, 48$
- Differences between adjacent numbers

  $1, 4, 4, 9, 5, 1, 6, 14, 1, 3$
- Differences for a high-frequency word (thus appearing in consecutive documents) are easier to compress, e.g.,

  $1, 1, 2, 1, 5, 1, 4, 1, 1, 3, \ldots$
- Differences for a low-frequency word are large, e.g., $109, 3766, 453, 1867, 992, \ldots$

Bit-Aligned Codes

- Breaks between encoded numbers can occur after any bit position
- *Unary* code
  - Encode k by k 1s followed by 0
  - 0 at end makes code unambiguous

Number	Code
0	0
1	10
2	110
3	1110
4	11110
5	111110

Unary and Binary Codes

- Unary is very efficient for small numbers such as 0 and 1, but quickly becomes very expensive
  - 1023 can be represented in 10 binary bits, but requires 1024 bits in unary
- Binary is more efficient for large numbers, but it may be ambiguous

Elias-γ Code

- To encode a number *k*, compute
  - $k_d = \lfloor \log_2 k \rfloor$
  - $k_r = k - 2^{\lfloor \log_2 k \rfloor}$
- k_d is number of binary digits, encoded in unary

Number (k)	k_d	k_r	Code
1	0	0	0
2	1	0	10 0
3	1	1	10 1
6	2	2	110 10
15	3	7	1110 111
16	4	0	11110 0000
255	7	127	11111110 1111111
1023	9	511	1111111110 111111111

Elias-γ Code and Decode

- For any number *k*, the Elias-γ code requires $\lfloor \log_2 k \rfloor + 1$ bits for k_d in unary code and $\lfloor \log_2 k \rfloor$ bits for k_r in binary code. A total of $2\lfloor \log_2 k \rfloor + 1$ bits are needed.
- When decoding, $k = 2^{kd} + k_r$
- For example, Elias-γ code = 1110110, k_d=*111*, k_r=*110*, *thus* $k = 2^3+6 = 14$
- Compared to binary coding, the saving comes from variable length coding.

63

Elias-δ Code

- Elias-γ code uses no more bits than unary, many fewer for k > 2
  - 1023 takes 19 bits instead of 1024 bits using unary
- In general, takes $2\lfloor \log_2 k \rfloor + 1$ bits
- To improve coding of large numbers, use Elias-δ code
  - Instead of encoding k_d in unary, we encode k_d + *1* using Elias-γ
  - Takes approximately $2 \log_2 \log_2 k + \log_2 k$ bits

Elias-δ Code

- Split k_d into:

$$k_{dd} = \lfloor \log_2 (k_d + 1) \rfloor$$

$$k_{dr} = (k_d + 1) - 2^{k_{dd}}$$

  - encode k_{dd} in unary, k_{dr} in binary, and k_r in binary

Number (k)	k_d	k_r	k_{dd}	k_{dr}	Code
1	0	0	0	0	0
2	1	0	1	0	10 0 0
3	1	1	1	0	10 0 1
6	2	2	1	1	10 1 10
15	3	7	2	0	110 00 111
16	4	0	2	1	110 01 0000
255	7	127	3	0	1110 000 1111111
1023	9	511	3	2	1110 010 111111111

Example of Elias-δ Code

- Code = 11000111
- From the code we know k_{dd} =*110*, k_{dr}=*00*, k_r=*111*
- $k = 2^{kd} + k_r$ *where* $k_d = k_{dr}+2^{kdd} -1$
- $k = 2^{0+4-1}+7 = 15$

```
#
# Generating Elias-gamma and Elias-delta codes in Python
#

import math

def unary_encode(n):
 return "1" * n + "0"

def binary_encode(n, width):
    r = ""
    for i in range(0,width):
     if ((1<<i) & n) > 0:
     r = "1" + r
     else:
      r = "0" + r
     return r

def gamma_encode(n):
    logn = int(math.log(n,2))
    return unary_encode( logn ) + " " + binary_encode(n, logn)

def delta_encode(n):
 logn = int(math.log(n,2))
if n == 1:
 return "0"
 else:
 loglog = int(math.log(logn+1,2))
 residual = logn+1 - int(math.pow(2, loglog))
        return unary_encode( loglog ) + " " + binary_encode( residual, loglog ) + " " + binary_encode(n, logn)

if __name__ == "__main__":
    for n in [1,2,3, 6, 15,16,255,1023]:
        logn = int(math.log(n,2))
        loglogn = int(math.log(logn+1,2))
        print n, "d_r", logn
        print n, "d_dd", loglogn
        print n, "d_dr", logn + 1 - int(math.pow(2,loglogn))
        print n, "delta", delta_encode(n)
        #print n, "gamma", gamma_encode(n)
        #print n, "binary", binary_encode(n)
```

Byte-Aligned Codes

- Variable-length bit encodings can be a problem on processors that process bytes
- *v-byte* is a popular byte-aligned code
  - Similar to Unicode UTF-8
- Shortest v-byte code is 1 byte
- Numbers are 1 to 4 bytes, with high bit 1 in the last byte, 0 otherwise

V-Byte Encoding

k	Number of bytes
$k < 2^7$	1
$2^7 \leq k < 2^{14}$	2
$2^{14} \leq k < 2^{21}$	3
$2^{21} \leq k < 2^{28}$	4

k	Binary Code	Hexadecimal
1	1 0000001	81
6	1 0000110	86
127	1 1111111	FF
128	0 0000001 1 0000000	01 80
130	0 0000001 1 0000010	01 82
20000	0 0000001 0 0011100 1 0100000	01 1C A0

V-Byte Encoder

```
public void encode( int[] input, ByteBuffer output ) {
    for( int i : input ) {
        while( i >= 128 ) {
            output.put( i & 0x7F );
            i >>>= 7;
        }
        output.put( i | 0x80 );
    }
}
```

V-Byte Decoder

```
public void decode( byte[] input, IntBuffer output ) {
    for( int i=0; i < input.length; i++ ) {
        int position = 0;
        int result = ((int)input[i] & 0x7F);

        while( (input[i] & 0x80) == 0 ) {
            i += 1;
            position += 1;
            int unsignedByte = ((int)input[i] & 0x7F);
            result |= (unsignedByte << (7*position));
        }

        output.put(result);
    }
}
```

Compression Example

- Consider invert list with positions:

$$(1, 2, [1, 7])(2, 3, [6, 17, 197])(3, 1, [1])$$

- Delta encode document numbers and positions:

$$(1, 2, [1, 6])(1, 3, [6, 11, 180])(1, 1, [1])$$

- Compress using v-byte:

```
81 82 81 86 81 82 86 8B 01 B4 81 81 81
```

Recap

- Index compression:
  - Delta coding
  - Elias-gamma
  - Elias-delta
  - Byte-aligned
    - V-Byte

Compression of Reuters

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
~, with blocking, k = 4	7.1
~, with blocking & front coding	5.9
collection (text, xml markup etc)	3600.0
collection (text)	960.0
T/D incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20-bits words)	250.0
postings, variable byte encoded	116.0
postings, γ encoded	101.0

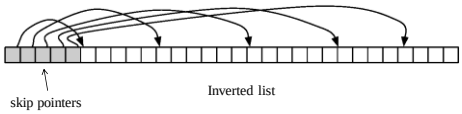
74

Skipping

- Search involves comparison of inverted lists of different lengths
  - Can be very inefficient
  - "Skipping" ahead to check document numbers is much better
  - Compression makes this difficult
    - Variable size, only d-gaps stored
- Skip pointers are additional data structure to support skipping

Skip Pointers

- A skip pointer *(d, p)* contains a document number *d* and a byte (or bit) position *p*
  - Means there is an inverted list posting that starts at position *p*, and the posting before it was for document *d*



Skip Pointers

- Example
  - Inverted list

$$5, 11, 17, 21, 26, 34, 36, 37, 45, 48, 51, 52, 57, 80, 89, 91, 94, 101, 104, 119$$

  - D-gaps

$$5, 6, 6, 4, 5, 9, 2, 1, 8, 3, 3, 1, 5, 23, 9, 2, 3, 7, 3, 15$$

  - Skip pointers

$$(17, 3), (34, 6), (45, 9), (52, 12), (89, 15), (101, 18)$$

Auxiliary Structures

- Inverted lists usually stored together in a single file for efficiency
  - *Inverted file*
- *Vocabulary* or *lexicon*
  - Contains a lookup table from index terms to the byte offset of the inverted list in the inverted file
  - Either hash table in memory or B-tree for larger vocabularies
- Term statistics stored at start of inverted lists
- Collection statistics stored in separate file