Information Retrieval (CSD510)

Index Construction and Compression

February 3, 2024

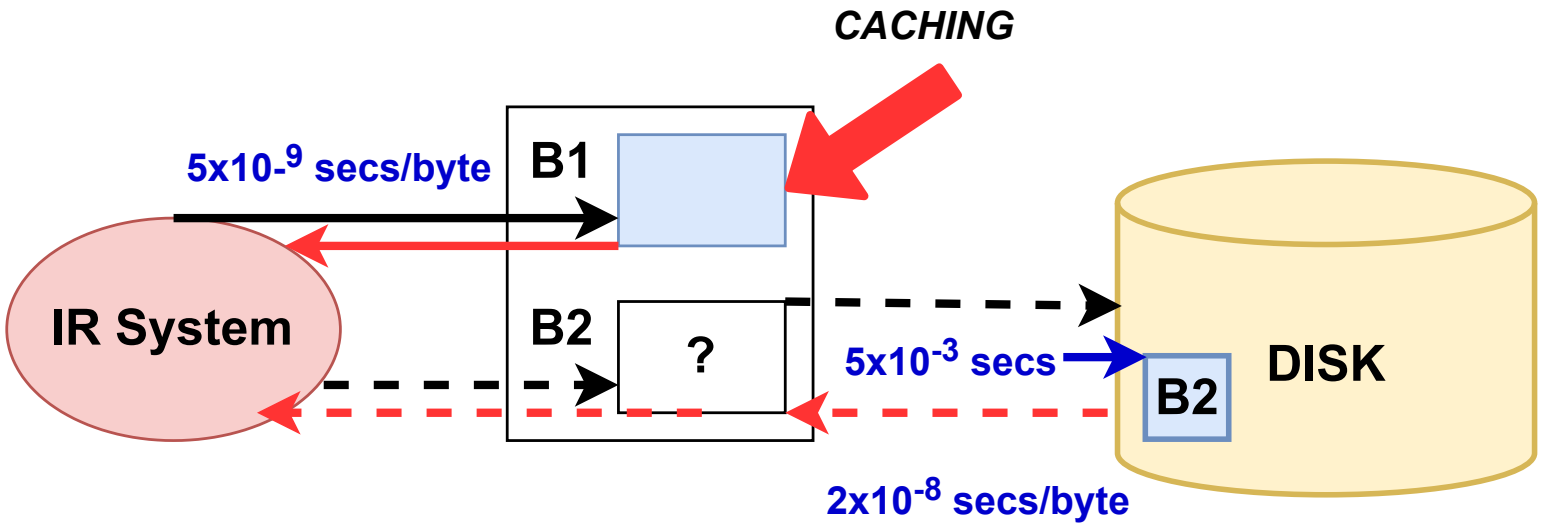
Lecture outline

Index Construction

Hardware Basics

- **When building an information retrieval (IR) system, many decisions are based on the characteristics of the computer hardware on which the system runs.**
- Access to data in memory is much faster than access to data on disk
  - **Caching** - Holding a part of data in main memory for faster access.
- Disk seeks
  - **Seek time:** Time required to locate the start of the data in the disk
  - No data transferred during seek time, slows down read process
  - Transferring one large chunk of data from disk to memory is faster than transferring many small chunks
- **Disk I/O is block-based**: Reading and writing of entire blocks as opposed to small chunks
  - **Block sizes:** 8KB to 256 KB.
- Servers typically have several gigabytes (GB) of main memory, sometimes hundreds of GB.
- Available disk space is in terabytes (TB)

Hardware basics



- Example: Read 10 MB data from disk
  - **Contiguous:** $(5 \times 10^{-3}) + (2 \times 10^{-8} \times 10 \times 10^{6}) \approx 0.2$ s
  - **100 non-contiguous blocks:**
    $(100 \times 5 \times 10^{-3}) + (2 \times 10^{-8} \times 10 \times 10^{6}) = 0.7$ s
- Chunks of related data to be stored in *contiguous chunks*.
- Data read in *blocks* rather than *bytes*.

Data set

- *Reuters-RCV1*
- English newswire articles published in a 12 month period (1995/6)


REUTERS

You are here: Home > News > Science > Article

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

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprints

[-] 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 meteorological base at Mawson Station on July 25.

statistic	**value**
# of documents (N)	8×10^5
avg. # tokens/doc (L)	200
# of terms (M)	4×10^5
avg. # bytes/token (incl. space, punct.)	6
avg. # bytes/token (without space, punct.)	6
avg. # bytes/term	7.5
non-positional postings	10^8

Inverted index

Consider the following documents

Doc 1: Breakthrough vaccine for Covid

Doc 2: New Covid vaccine

Doc 3: A new approach to vaccination against Covid

Doc 4: New hopes for Covid patients

- **Tokens:** Breakthrough, vaccine, for, Covid, New, A, new, approach, to, vaccination, against, hopes, patients
- **Case normalization:** breakthrough, vaccine, for, covid, a, new, approach, to, vaccination, against, hopes, patients
- **Stopword removal** breakthrough, vaccine, covid, new, approach, vaccination, against, hopes, patients (a, for, to)
- **Stemming:** breakthrough, **vaccin**, covid, new, approach, against, **hope**, **patient**
- **Index terms:** breakthrough, vaccin, covid, new, approach, against, hope, patient

Inverted index

Sort by **docID**

breakthrough	1
vaccin	1
covid	1
new	2
covid	2
vaccin	2
new	3
approach	3
vaccin	3
against	3
covid	3
new	4
hope	4
covid	4
patient	4

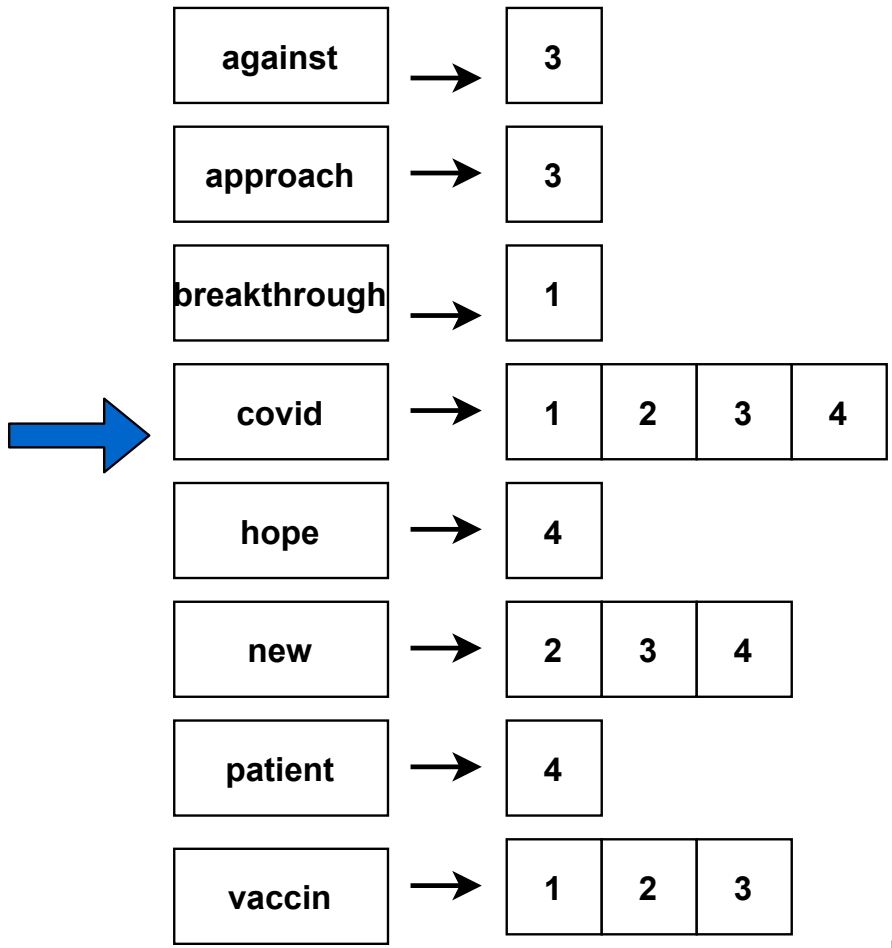
Sort by **terms**

against	3
approach	3
breakthrough	1
covid	1
covid	2
covid	3
covid	4
hope	4
new	2
new	3
new	4
patient	4
vaccin	1
vaccin	2
vaccin	3

Building Inverted Index

- Multiple term entries in a single document are merged.
- Split into Dictionary and Postings
- Document frequency information is added to dictionary entries.

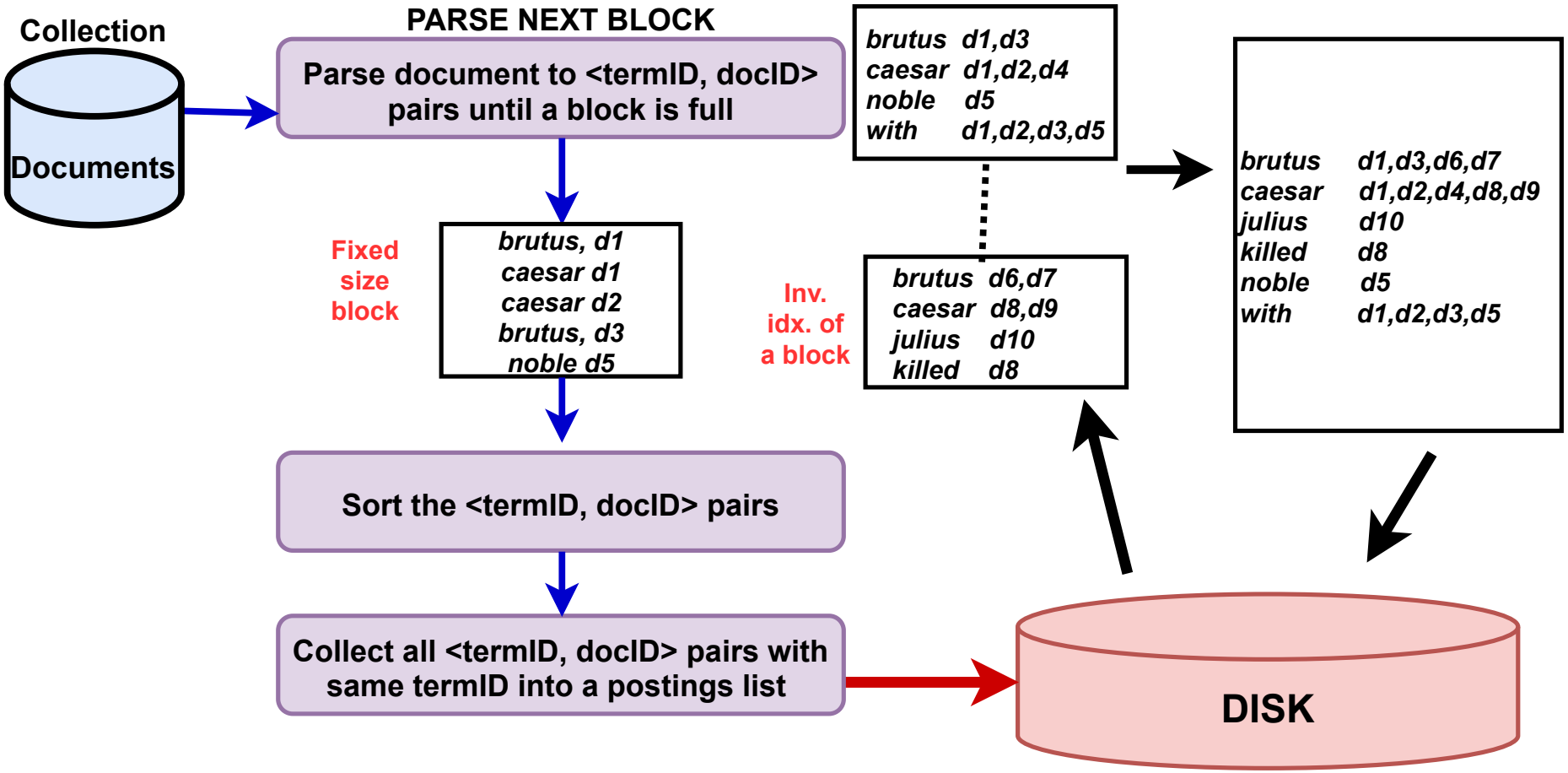
against	3
approach	3
breakthrough	1
covid	1
covid	2
covid	3
covid	4
hope	4
new	2
new	3
new	4
patient	4
vaccin	1
vaccin	2
vaccin	3



Sort-based indexing

- As we build index, we parse docs one at a time
  - Generate the (term/termID, docID) pairs
- At the end, sort the entire collection with term/termID as first key and docID as secondary key.
- We cannot keep all postings in memory and do the sort in-memory for large collections
- We need to use the disk along with the main memory for sorting.
- We may store intermediate results on disk.
- We need an **external sorting algorithm**.
  1. Blocked sort-based indexing (BSBI)
  2. Single pass in-memory indexing (SPIMI)
- **Requirement:** **Minimize** the number of **random disk seeks**

Blocked sort-based Indexing (BSBI)



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)
```

- **PARSENEXTBLOCK**
  1. Parse documents into termID-docID pairs
  2. Accumulate the pairs in memory until a block of a fixed size is full
  3. Sort the termID-docID
  4. Collect all termID–docID pairs with the same termID into a postings list
- A *posting* is simply a docID.
- Block size chosen to **fit in memory** and **enable fast in-memory sort**

- **MERGEBLOCKS**
  1. Open all blocks simultaneously
  2. Maintain small buffer for each block
  3. In each iteration, choose the lowest *termID*
  4. Write back the merged inverted index to disk

Blocked sort-based Indexing (BSBI)

- BSBI requires an algorithm to map **terms** to **termIDs**.
- Data structure mapping **terms** to **termIDs** may be too large to fit in memory.
- The **sorting step** may be expensive.
- **Scalable alternative: Single pass in-memory indexing (SPIMI)**

Single pass in-memory indexing (SPIMI)

- SPIMI uses terms instead of termIDs - no need to maintain *term to termID mapping*.
- Dynamically constructs dictionary eliminating expensive sorting.
- Can index collections of any size as long as there is enough disk space available.
- Compression makes SPIMI even more efficient

Single pass in-memory indexing (SPIMI)

- **Steps:**
  1. PARSE each document to generate *token_stream* consisting of $< term, docID >$ pairs

```
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
```

  2.
  3. MERGE the blocks to a single **inverted index** by a linear scan
- Sort the **dictionary** in each *block* in lexicographic order before writing to facilitate merge by **linear scan**

Distributed indexing

- Data Collection is too big that index construction cannot be performed efficiently on a single machine
- A **distributed computing cluster** must be used for web-scale indexing
- Web search engines use **distributed indexing algorithms** for index construction
- Distributed index partitioned across several machines according to
  1. **terms** - A machine contains part of index for a range of terms
  2. **documents** - A machine contains part of index for a subset of documents

Distributed indexing

- Large computer *clusters* to construct any reasonably sized web index.
- **Clustering** is to solve large computing problems on cheap commodity machines (nodes)
- Individual machines are fault-prone - Can unpredictably slow down or fail
- **Solution:**
  1. Divide task into *smaller subtasks.*
  2. *Assign* a subtask to a machine
  3. *Reassign* the subtask in case of failure

Distributed indexing - *MapReduce*

- **MapReduce** is a general architecture for distributed computing
- MapReduce is designed for large computer clusters
- Distributed indexing uses the **MapReduce** framework
- It comprises of a **master node** and rest all are **worker nodes**
- A **master node** directs the process of assigning and reassigning tasks to individual **worker machines**.
- The **map** and **reduce** phases of *MapReduce* split up computing job into subtasks to be solved by worker nodes.
- **Indexing using MapReduce**
  - Break up indexing into sets of parallel tasks
  - Master machine assigns each task to an idle machine from a pool

Distributed indexing

- Two sets of parallel tasks defined for indexing:
  - Parsers (Map phase)
  - Indexers (Reduce phase)
- Break the input document collection into **splits**
- Each split is a subset of documents

Distributed indexing - Parser (*Map* phase)

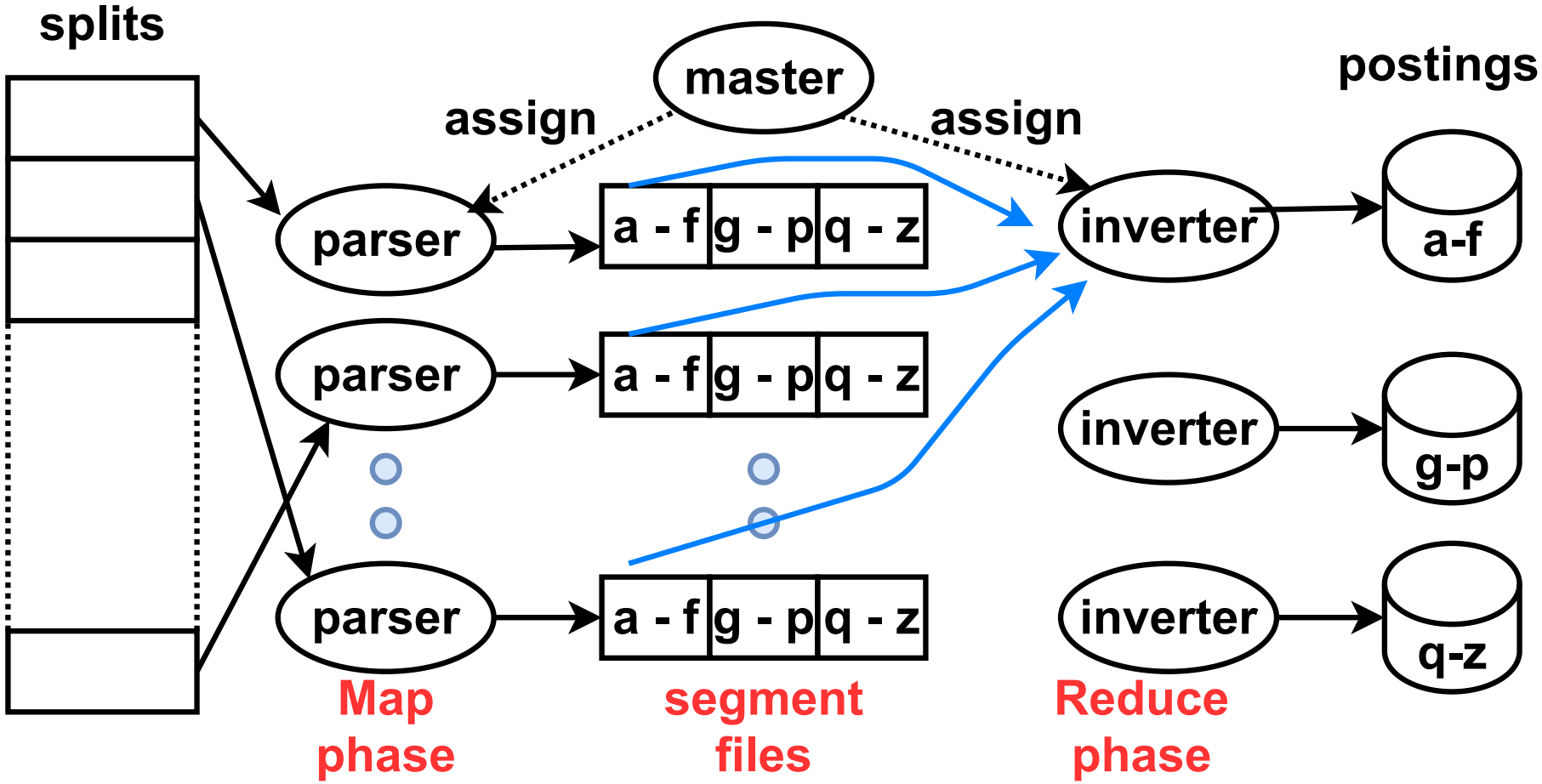
- Master assigns a split to an idle parser machine.
- **Parser** reads a document at a time and emits (term,docID)-pairs.
- Parser writes pairs into j **term-partitions**.
- Each for a **range** of terms' *first letters*

  e.g., a-f, g-p, q-z (here: j = 3)
- Each partition is written to a separate **segment file**.

Distributed indexing - Indexer (Reduce phase)

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

Distributed indexing



Distributed indexing

- MapReduce is a robust and conceptually simple framework for distributed computing ...
- ... without having to write code for the distribution part
- The Google indexing system consists of a number of phases, each implemented in MapReduce.
- Index construction was just one phase.
- Another phase: transform term-partitioned into **document-partitioned index.**

Dynamic indexing

- Up to now, we have assumed that collections are static.
- Documents are rarely *inserted, deleted and modified* in the collection
- Otherwise, it implies that the dictionary and postings lists have to be dynamically modified.

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
- Complexity
  - T - Total size of index (seen so far)
  - n - Size of an auxiliary index
  - T/n - # of merges
  - Each posting is touched once during *each* merge operation
  - **Complexity:** $O(T^2/n)$
- Issue with auxiliary and main index
  - Frequent merges
  - Poor search performance during index merge

Dynamic indexing: Logarithmic merge

- Reduces the merging cost
- Reduction achieved at the cost of search time
- Maintain a series of indexes, each twice as large as the previous one.
- Keep smallest index of size (Z_0) in memory
- Keep larger indices ($I_0, I_1, \cdots$) in disk.
- If Z_0 gets too big ($> n$),
  - write to disk as I_0
  - or, if I_0 exists, merge with I_0 and write the merger as I_1 etc.

Dynamic indexing: Logarithmic merge

```
LMergeAddToken(indexes, Z_0, token)
 1  Z_0 ← Merge(Z_0, {token})
 2  if |Z_0| = n
 3     then for i ← 0 to ∞
 4          do if I_i ∈ indexes
 5                then Z_{i+1} ← Merge(I_i, Z_i)
 6                     (Z_{i+1} is a temporary index on disk.)
 7                     indexes ← indexes − {I_i}
 8                else I_i ← Z_i     (Z_i becomes the permanent index I_i.)
 9                     indexes ← indexes ∪ {I_i}
10                     Break
11          Z_0 ← ∅
```

```
LogarithmicMerge()
1  Z_0 ← ∅     (Z_0 is the in-memory index.)
2  indexes ← ∅
3  while true
4  do LMergeAddToken(indexes, Z_0, getNextToken())
```

Binary numbers: $I_2 I_1 I_0 = 2^2 2^1 2^0$

- 001
- 010
- 011
- 100
- 101
- 110
- 111

Dynamic indexing: Logarithmic merge

- At any point of time, there are $log_2(T/n)$ indexes $(I_0, I_1, I_2, \cdots)$ of sizes $2^0 \times n, 2^1 \times n, 2^2 \times n, \cdots$
- Postings percolate up this sequence of indices and processed once at each stage
- Complexity
  - $\Theta(Tlog_2(T/n))$
  - # of levels: $log_2(T/n)$
  - Each of the T postings touched once during each merge.

Compression and vocabulary basics

Why care to do compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Increase speed of transferring data from disk to memory (again, increases speed)
  - reading compressed data and decompress in memory
    is faster than
    read uncompressed data
- Premise: Decompression algorithms are fast.
- This is true of the decompression algorithms we will use.

Why compression in information retrieval?

- Space for **dictionary**
  - Main motivation for dictionary compression: make it small enough to keep in main memory
- Space for **the postings file**
  - Motivation: reduce disk space needed, decrease time needed to read from disk
  - Note: Large search engines keep significant part of postings in memory
- We shall explore various compression schemes for dictionary and postings.

Lossy vs. lossless compression

- **Lossy compression**: Discards some information that cannot be recovered later.
- Several of the preprocessing steps we frequently use can be viewed as lossy compression
  - casing-folding, stop words removal, stemming, number elimination
- **Lossless compression**: All information is preserved.
  - What is mostly done in index compression

How big is the term vocabulary?

- That is, how many distinct words are there?
- Can we assume there is an upper bound?

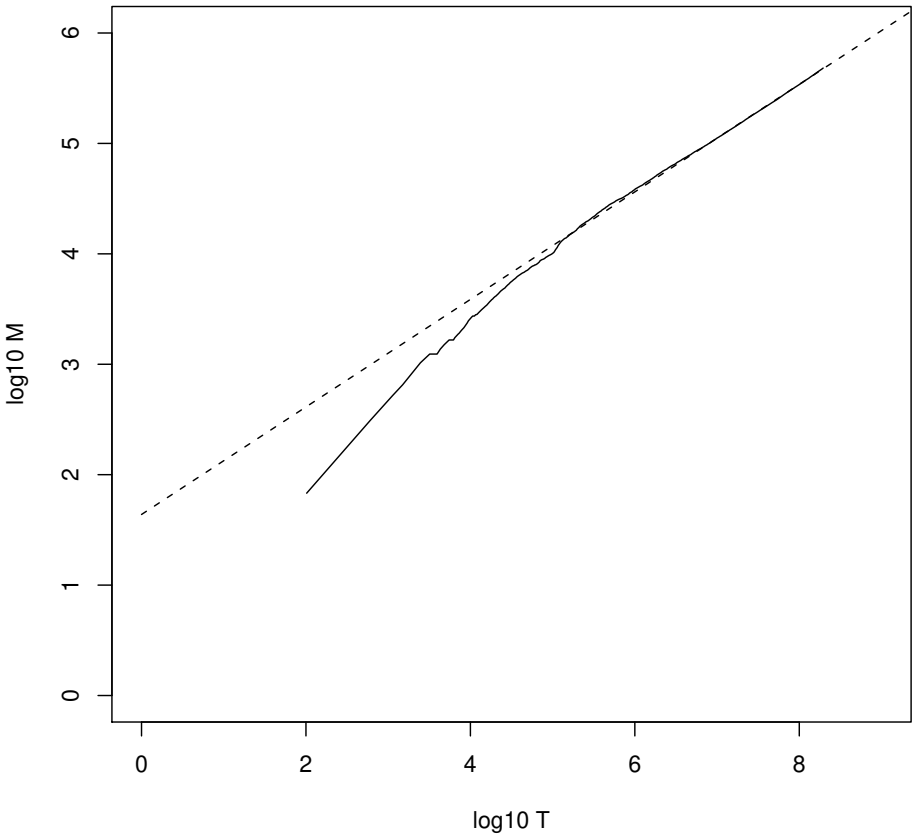
How big is the term vocabulary?

- That is, how many distinct words are there?
- Can we assume there is an upper bound?
- **Not really !**
- The vocabulary will keep growing with collection size

How big is the term vocabulary?

- That is, how many distinct words are there?
- Can we assume there is an upper bound?
- **Not really !**
- The vocabulary will keep growing with collection size
- **Heaps' law**: $M = kT^b$
- M is the size of the vocabulary, T is the number of tokens in the collection.
- Typical values for the parameters k and b are: $30 \leq k \leq 100$ and $b \approx 0.5$
- Heaps' law is linear in log-log space
  - Empirical law
  - It is the simplest possible relationship between **collection size** and **vocabulary size** in *log-log space*

Heaps' law for Reuters



- Vocabulary size M as a function of collection size T (number of tokens) f Reuters-RCV1
- The dashed line is the best least squares fit:

  $log_{10} M = 0.49 log_{10} T + 1.64$
- Thus, $M = 10^{1.64} T^{0.49}$, $k = 10^{1.64} \approx$ and $b = 0.49$

- Example: for the first 1,000,020 tokens Heaps' law predicts 38,323 terms:

$$44 \times 1,000,020^{0.49} \approx 38,323$$

- The actual number is 38,365 terms, very close to the prediction

Zipf's law

- Estimate how many **frequent vs. infrequent terms** we should *expect* in a collection.
- In natural language, there are a few very frequent terms and many very rare terms.
- **Zipf's law:** The i^{th} most frequent term has frequency cf_i proportional to $\frac{1}{i}$

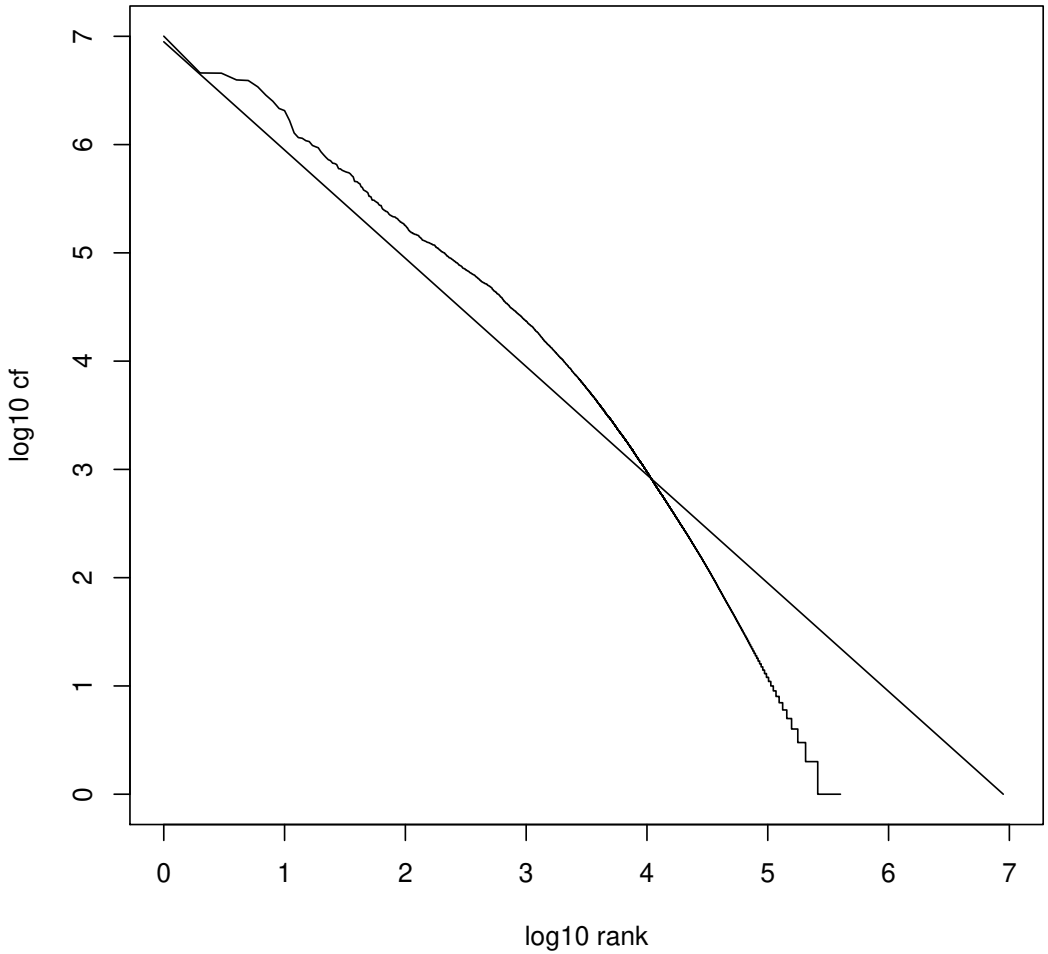
$$cf_i \propto \frac{1}{i}$$

- cf_i is collection frequency: the number of occurrences of the term t_i in the collection.

Zipf's law

- So if the most frequent term (*the*) occurs cf_1 times, then the second most frequent term (*of*) has half as many occurrences $cf_2 = \frac{1}{2}cf_1 \cdots$
- $\cdots$ and the third most frequent term (*and*) has a third as many occurrences $cf_3 = \frac{1}{3}cf_1$ etc.
- Equivalent: $cf_i = ci^k$ and $log\ cf_i = log\ c + k\ log\ i$ ($for\ k = -1$)
- Example of a **power law**

Zipf's law for Reuters



- **Key insight:** Few frequent terms, many rare terms.

Dictionary compression

Dictionary compression

- The dictionary is small compared to the postings file.
- But we want to keep it in memory.
- Also: competition with other applications, cell phones, onboard computers, fast startup time
- So compressing the dictionary is important.

Recall: Dictionary as array of fixed-width entries

term	document frequency	pointer to postings list
a	656,265	$\longrightarrow$
aachen	65	$\longrightarrow$
...	...	...
zulu	221	$\longrightarrow$
space needed:	20 bytes	4 bytes	4 bytes

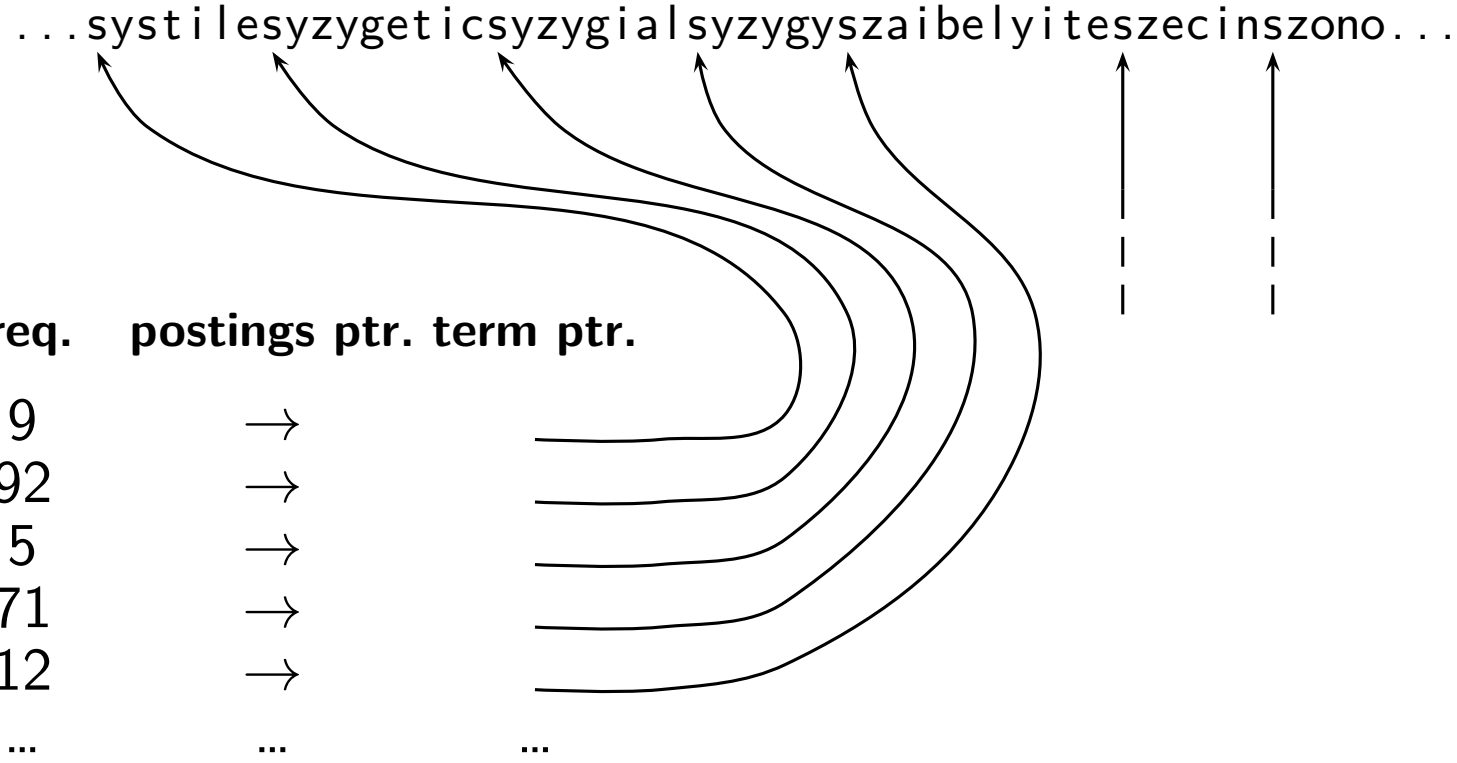
Space for

Reuters: (20+4+4)*400,000 = **11.2 MB**

Cons

- Most of the bytes in the term column are wasted.
  - We allot 20 bytes for terms of length 1.
- We can't handle long terms like `hydrochlorofluorocarbons` (24 chs)
- Average length of a term in English: 8 characters (or a little bit less)
- How can we use on average 8 characters per term?

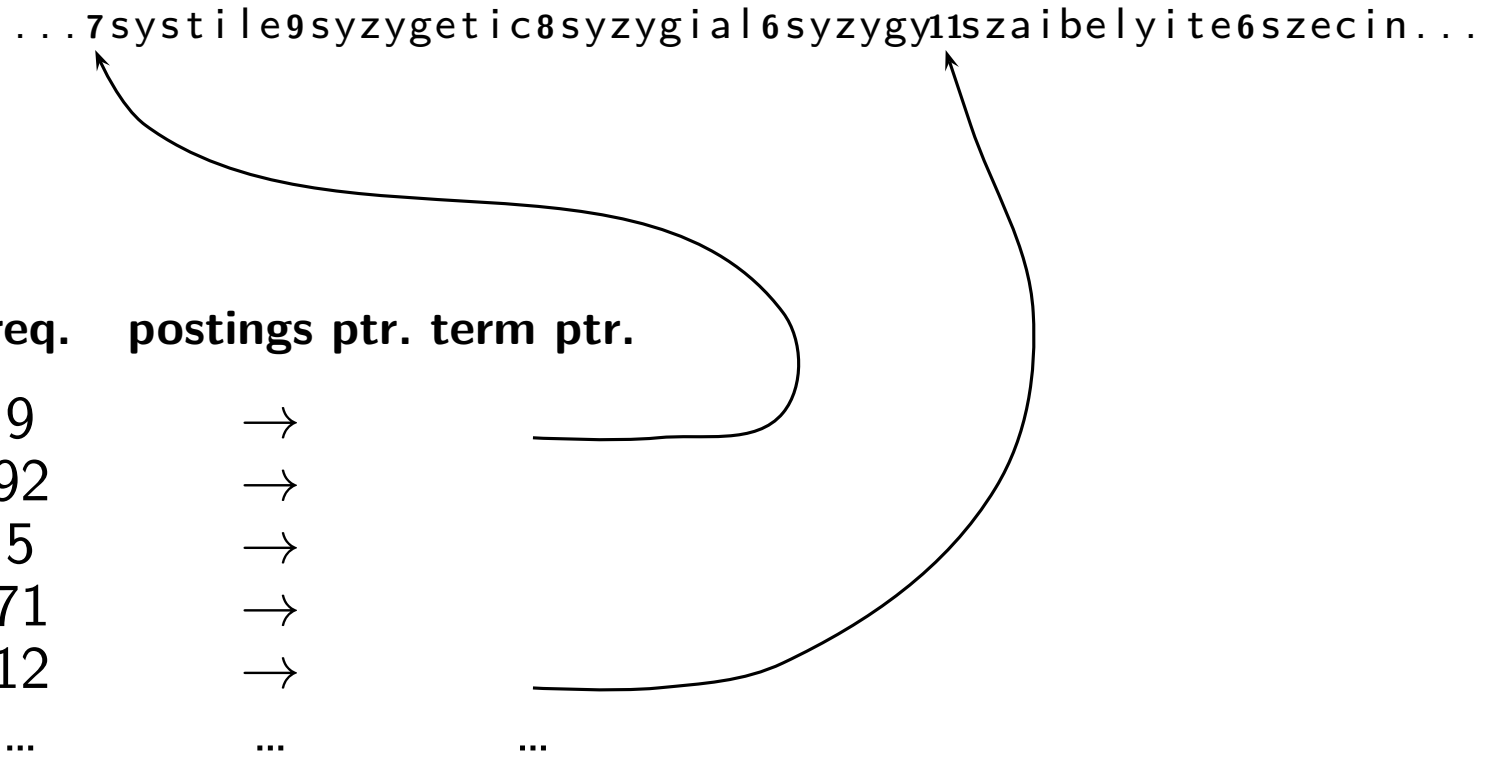
Dictionary as a string



4 bytes 4 bytes 3 bytes

- 4 bytes per term for frequency
- 4 bytes per term for pointer to postings list
- 8 bytes (on average) for term in string
- 3 bytes per pointer into string (need $\log_2 8 \cdot 400000 < 24$ bits to resolve $8 \cdot 400{,}000$ positions)
- Space: $400000 \times (4 + 4 + 3 + 8) = \mathbf{7.6}$ MB (compared to 11.2 MB)

Dictionary as a string with blocking

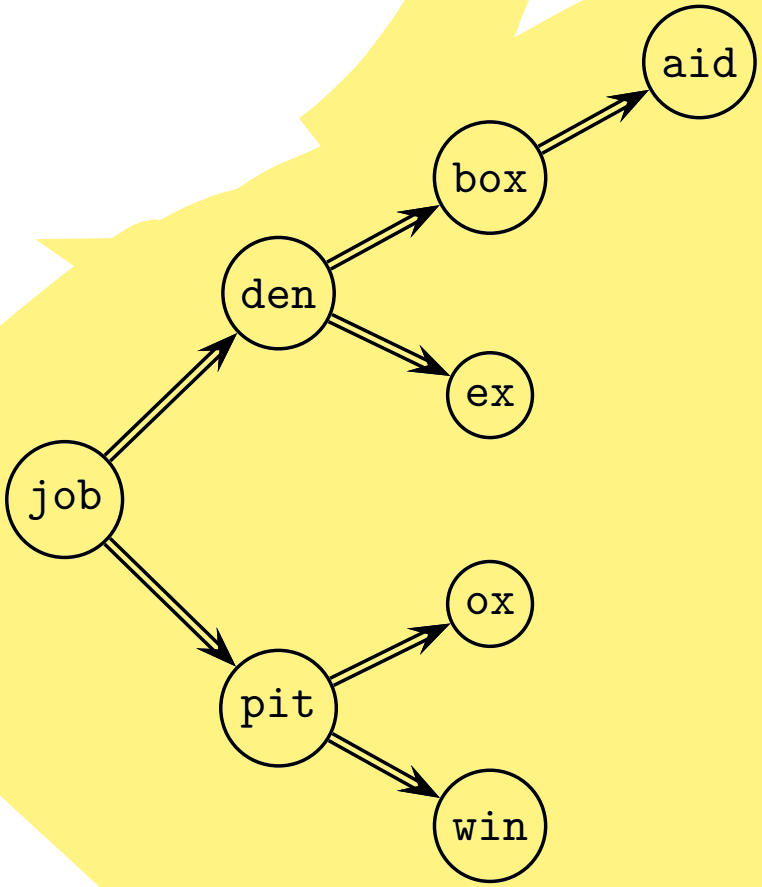


Space for dictionary as a string with blocking

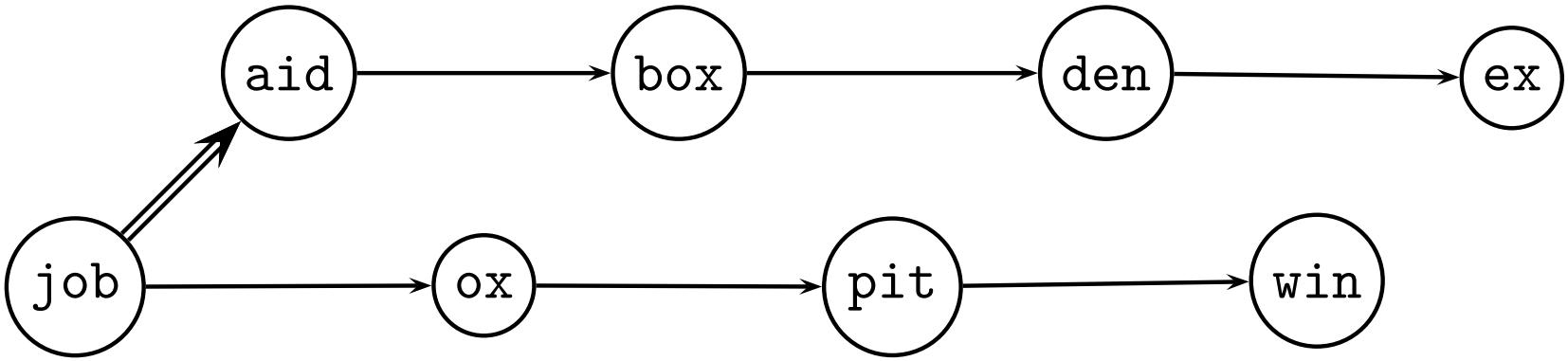
- Example block size $k = 4$
- Where we used 4×3 bytes for term pointers without blocking

  we now use 3 bytes for one pointer plus 4 bytes for indicating the length of each term.
- We save $12 - (3 + 4) = 5$ bytes per block
- Total savings: $400{,}000/4 * 5 = 0.5$ MB
- This reduces the size of the dictionary from 7.6 MB to 7.1 MB

Lookup of a term without blocking



Lookup of a term with blocking: (slightly) slower



Front coding

One block in blocked compression ($k = 4$) ...

8 a u t o m a t a **8** a u t o m a t e **9** a u t o m a t i c **10** a u t o m a t i o n

$\Downarrow$

...further compressed with front coding.

8 a u t o m a t $*$ a **1** $\diamond$ e **2** $\diamond$ i c **3** $\diamond$ i o n

Dictionary compression for Reuters: Summary

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
$\sim$, with blocking, $k = 4$	7.1
$\sim$, with blocking & front coding	5.9

Postings file compression

Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- **Purpose:** Store each posting compactly
- A posting for our purposes is a docID.
- For Reuters (800,000 documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use $log_2 800{,}000 \approx 19.6 < 20$ bits per docID
- *Objective:* use a lot less than 20 bits per docID

Postings compression (docID gap-based compression)

- In a postings list, postings are sorted in increasing order of docID.
- `COMPUTER`: 283154, 283159, 283202, $\cdots$
- It suffices to store **gaps** instead of the actual values.

  283159-283154=5, 283202-283159=43
- Example postings list using gaps :

  `COMPUTER`: 283154, 5, 43, $\cdots$
- Gaps for frequent terms are small.
- We can encode small gaps with fewer than 20 bits.

Gap encoding

	encoding	postings list								
the	docIDs	...		283042		283043		283044		283045	...
	gaps				1		1		1		...
computer	docIDs	...		283047		283154		283159		283202	...
	gaps				107		5		43		...
arachnocentric	docIDs	252000		500100							
	gaps	252000	248100								

Variable length encoding

Aim

- For `ARACHNOCENTRIC` and other rare terms, we will use about 20 bits per gap (= posting).
- For `THE` and other very frequent terms, we will use only a few bits per gap (= posting).

- We need a variable length encoding to implement this.
- Variable length encoding uses few bits for small gaps and many bits for large gaps.

Variable byte (VB) code

- Used by many commercial/research systems
- Dedicate 1 bit (high bit) to be a continuation bit c.
- If the gap G fits within 7 bits, binary-encode it in the 7 available bits and set $c = 1$.
- Else: encode lower-order 7 bits and then use one or more additional bytes to encode the higher order bits using the same algorithm.
- At the end set the continuation bit of the last byte to 1 ($c = 1$) and of the other bytes to 0 ($c = 0$)

VB code examples

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

VB code encoding algorithm

```
VBEncodeNumber(n)
1  bytes ← ⟨⟩
2  while true
3  do Prepend(bytes, n mod 128)
4     if n < 128
5        then Break
6     n ← n div 128
7  bytes[Length(bytes)] += 128
8  return bytes
```

```
VBDecode(bytestream)
1  numbers ← ⟨⟩
2  n ← 0
3  for i ← 1 to Length(bytestream)
4  do if bytestream[i] < 128
5        then n ← 128 × n + bytestream[i]
6        else n ← 128 × n + (bytestream[i] − 128)
7             Append(numbers, n)
8             n ← 0
9  return numbers
```

```
VBEncode(numbers)
1  bytestream ← ⟨⟩
2  for each n ∈ numbers
3  do bytes ← VBEncodeNumber(n)
4     bytestream ← Extend(bytestream, bytes)
5  return bytestream
```

Other variable codes

- Instead of bytes, we can also use a different "unit of alignment": 32 bits (words), 16 bits, 4 bits (nibbles) etc
- Variable byte alignment wastes space if you have many small gaps – nibbles do better on those
- There is work on word-aligned codes that efficiently "pack" a variable number of gaps into one *word*

Gamma codes for gap encoding

- We can achieve even more compression with another type of variable length encoding: bitlevel code
- **Gamma code** is the best known of these.
- First, we need to understand *unary code* to introduce *gamma code*
- **Unary code**
  - Represent n as n 1s with a final 0.
  - Unary code for 3 is 1110
  - Unary code for 40 is 11111111111111111111111111111111111110

Gamma code

- Represent a gap G as a pair of **length** and **offset**.
- Offset is the gap in binary, with the leading bit chopped off.
- *For example:* 14 $\rightarrow$ 1110 $\rightarrow$ 110
- Length is the length of offset.

  For 14 (offset 110), this is 3.

  Encode length in unary code: 1110
- Gamma code of 14 is the concatenation of length and offset: 1110110

Length of gamma code

- Length of *offset*: $\lfloor log_2 G \rfloor$ bits
- Length of *length*: $\lfloor log_2 G \rfloor + 1$ bits
- So the length of the entire code is $2 \times \lfloor log_2 G \rfloor + 1$ bits
- γ codes are
  - always of odd length
  - within a factor of 2 of the optimal encoding length $\lfloor log_2 G \rfloor$

Properties of γ code

- **prefix-free** No γ code is prefix of another
- **universal:** Within a factor of the optimal
- **parameter-free:** No parameter needs to be maintained for generating γ code

Compression of Reuters

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
$\sim$, with blocking, $k = 4$	7.1
$\sim$, 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)	250.0
postings, variable byte encoded	116.0
postings, γ encoded	101.0