

# Indexing

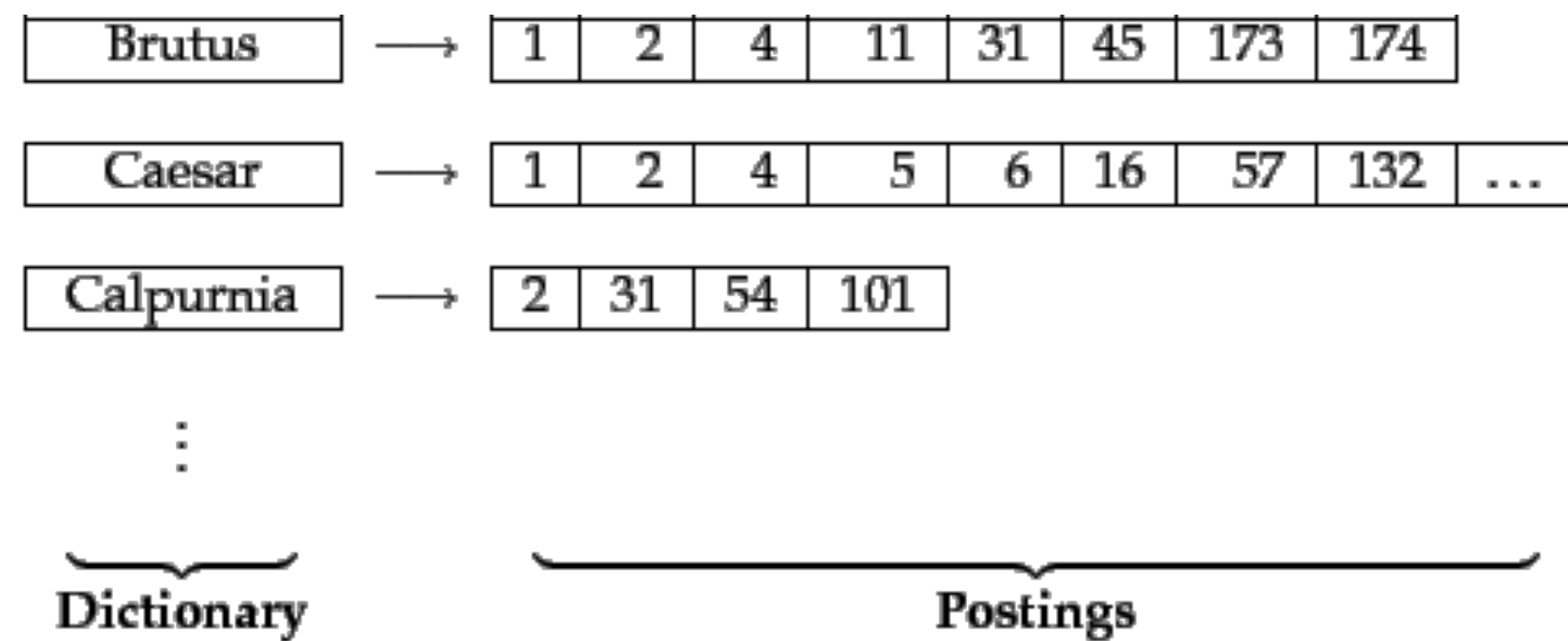
Index Construction

# Motivation: Scale

| Corpus              | Terms        | Docs         | Entries        |
|---------------------|--------------|--------------|----------------|
| Shakespeare's Plays | ~31,000      | 37           | ~1.1 million   |
| English Wikipedia   | ~1.7 million | ~4.5 million | ~7.65 trillion |
| English Web         | >2 million   | >1.7 billion | >3.4x10        |

- A term incidence matrix with  $V$  terms and  $D$  documents has  $O(V \times D)$  entries.
- Shakespeare used around 31,000 distinct words across 37 plays, for about 1.1M entries.
- As of 2014, a collection of Wikipedia pages comprises about 4.5M pages and roughly 1.7M distinct words. Assuming just one bit per matrix entry, this would consume about 890GB of memory.

# Inverted Indexes - Intro



► **Figure 1.2** The two parts of an inverted index. The dictionary is commonly kept in memory, with pointers to each postings list, which is stored on disk.

- Two insights allow us to reduce this to a manageable size:
  1. The matrix is *sparse* – any document uses a tiny fraction of the vocabulary.
  2. A query only uses a handful of words, so we don't need the rest.
- We use an *inverted index* instead of using a term incidence matrix directly.
- An inverted index is a map from a term to a *posting list* of documents which use that term.

# Search Algorithm

```
1 def runQuery([t1, t2, ..., tn]):
2     terms = sortByIncreasingFrequency([t1, t2, ..., tn])
3     result = terms[0].postings
4     for term in terms[1:]:
5         result = intersect(result, term.postings)
6     return result
7
8 def intersect(p1, p2):
9     answer = []
10    i = j = 0
11    while i < len(p1) and j < len(p2):
12        if p1[i] == p2[j]:
13            answer.add(p1[i])
14            i += 1
15            j += 1
16        elif p1[i] < p2[j]:
17            i += 1
18        else:
19            j += 1
20    return answer
```

- Consider queries of the form:  
 $t_1 \text{ AND } t_2 \text{ AND } \dots \text{ AND } t_n$
- In this simplified case, we need only take the intersections of the term posting lists.
- This algorithm, inspired by merge sort, relies on the posting lists being sorted by length.
- We save time by processing the terms in order from least common to most common. (Why does this help?)

# Motivation

---

- All modern search engines rely on inverted indexes in some form. Many other data structures were considered, but none has matched its efficiency.
- The entries in a production inverted index typically contain many more fields providing extra information about the documents.
- The efficient construction and use of inverted indexes is a topic of its own, and will be covered in a later module.

# Motivation

---

A reasonably-sized index of the web contains many billions of documents and has a massive vocabulary.

Search engines run roughly  $10^5$  queries per second over that collection.

We need fine-tuned data structures and algorithms to provide search results in much less than a second per query.  $O(n)$  and even  $O(\log n)$  algorithms are often not nearly fast enough.

The solution to this challenge is to run an inverted index on a massive distributed system.

# Inverted Indexes

Inverted Indexes are primarily used to allow fast, concurrent query processing.

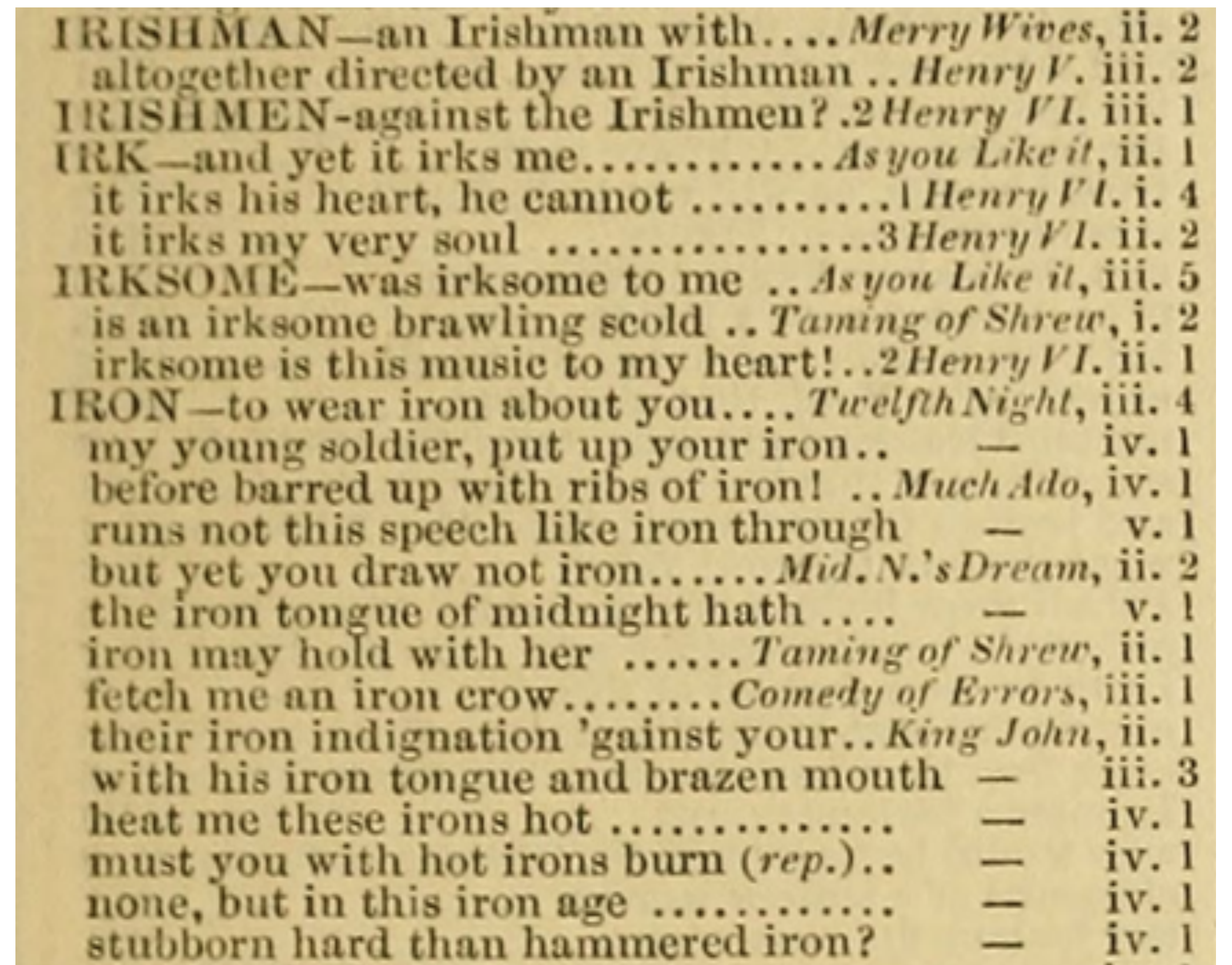
Each term found in any indexed document receives an independent inverted list, which stores the information necessary to process that term when it occurs in a query.

# Indexes

The primary purpose of a search engine index is to store whatever information is needed to minimize processing at query time.

Text search has unique needs compared to, e.g., database queries, and needs its own data structures – primarily, the inverted index.

- A **forward index** is a map from documents to terms (and positions). These are used when you search within a document.
- An **inverted index** is a map from terms to documents (and positions). These are used when you want to find a term in any document.



|          |  |                                  |
|----------|--|----------------------------------|
| IRISHMAN | —an Irishman with....                          | <i>Merry Wives</i> , ii. 2       |
|          | altogether directed by an Irishman ..          | <i>Henry V.</i> iii. 2           |
| IRISHMEN | —against the Irishmen? .2                      | <i>Henry VI.</i> iii. 1          |
| IRK      | —and yet it irks me.....                       | <i>As you Like it</i> , ii. 1    |
|          | it irks his heart, he cannot .....             | <i>Henry VI.</i> i. 4            |
|          | it irks my very soul .....                     | <i>Henry VI.</i> ii. 2           |
| IRKSOME  | —was irksome to me ..                          | <i>As you Like it</i> , iii. 5   |
|          | is an irksome brawling scold ..                | <i>Taming of Shrew</i> , i. 2    |
|          | irksome is this music to my heart!..           | <i>Henry VI.</i> ii. 1           |
| IRON     | —to wear iron about you....                    | <i>Twelfth Night</i> , iii. 4    |
|          | my young soldier, put up your iron..           | — iv. 1                          |
|          | before barred up with ribs of iron! ..         | <i>Much Ado</i> , iv. 1          |
|          | runs not this speech like iron through         | — v. 1                           |
|          | but yet you draw not iron.....                 | <i>Mid. N.'s Dream</i> , ii. 2   |
|          | the iron tongue of midnight hath ....          | — v. 1                           |
|          | iron may hold with her .....                   | <i>Taming of Shrew</i> , ii. 1   |
|          | fetch me an iron crow.....                     | <i>Comedy of Errors</i> , iii. 1 |
|          | their iron indignation 'gainst your..          | <i>King John</i> , ii. 1         |
|          | with his iron tongue and brazen mouth          | — iii. 3                         |
|          | heat me these irons hot .....                  | — iv. 1                          |
|          | must you with hot irons burn ( <i>rep.</i> ).. | — iv. 1                          |
|          | none, but in this iron age .....               | — iv. 1                          |
|          | stubborn hard than hammered iron?              | — iv. 1                          |

**Is this a forward or an inverted index?**

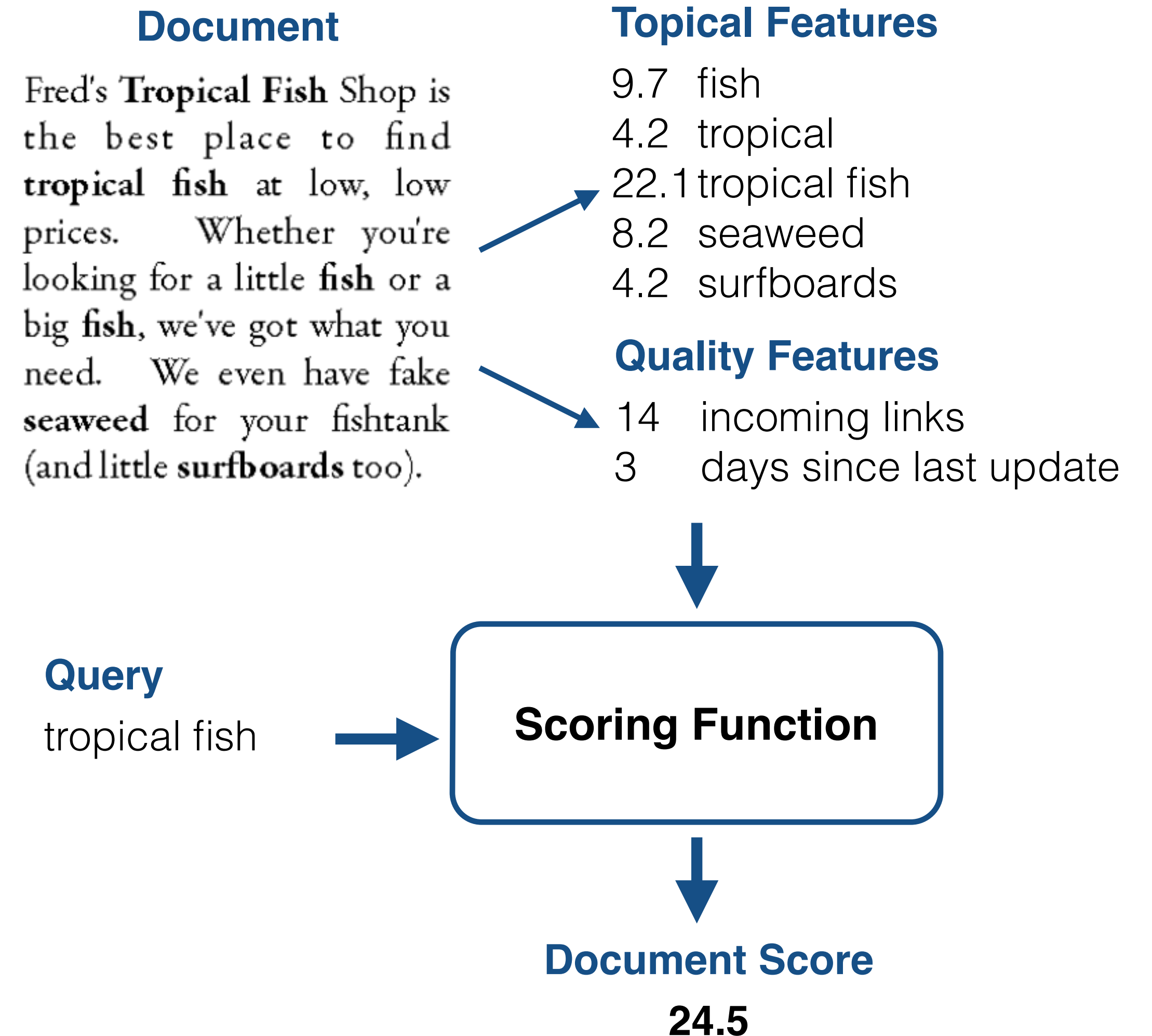


# Abstract Model of Ranking

Indexes are created to support search, and the primary search task is *document ranking*.

We sort documents according to some scoring function which depends on the terms in the query and the document representation.

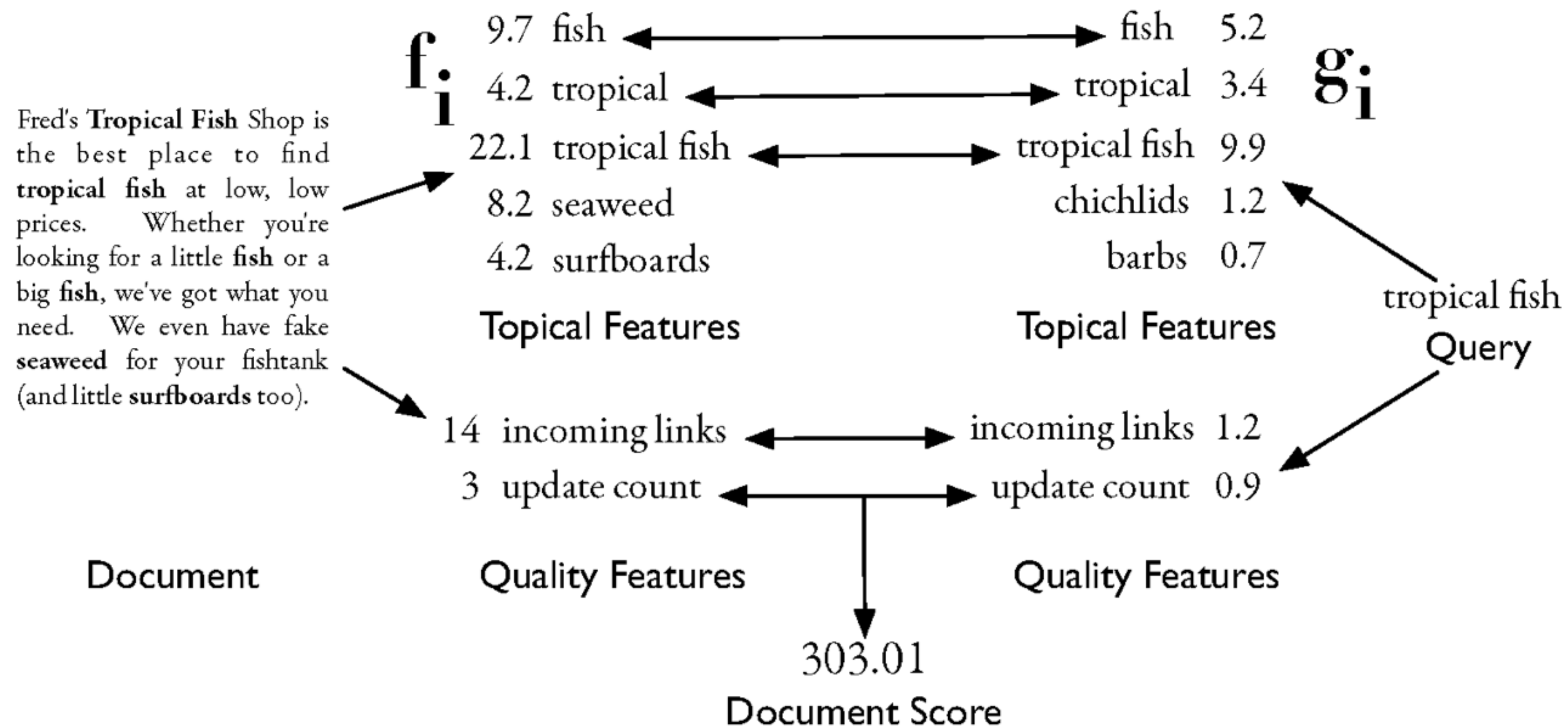
In the abstract, we need to store various document features to efficiently score documents in response to a query.



# More Concrete Model

$$R(Q, D) = \sum_i g_i(Q) f_i(D)$$

$f_i$  is a document feature function  
 $g_i$  is a query feature function

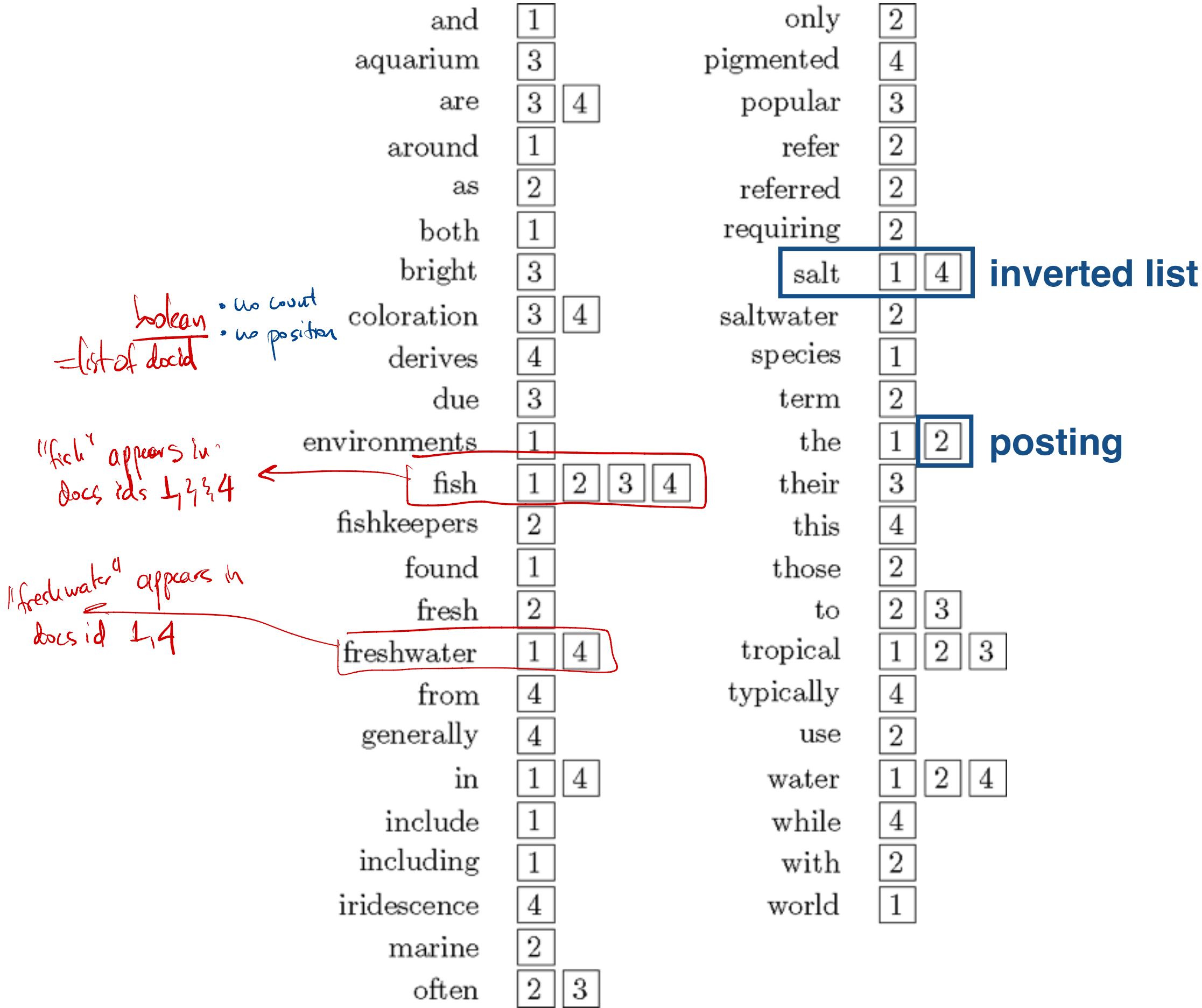


# Inverted Lists

In an inverted index, each term has an associated **inverted list**.

At minimum, this list contains a list of identifiers for documents which contain that term.

Usually we have more detailed information for each document as it relates to that term. Each entry in an inverted list is called a **posting**.



Simple Inverted Index

# Inverted Index with Counts

Document postings can store any information needed for efficient ranking.

For instance, they typically store term counts for each document –  $tf_{w,d}$ .

Depending on the underlying storage system, it can be expensive to increase the size of a posting. It's important to be able to efficiently scan through an inverted list, and it helps if they're small.

|              |     |     |     |     |  |           |             |
|--------------|-----|-----|-----|-----|--|-----------|-------------|
| and          | 1:1 |     |     |     |  | only      | 2:1         |
| aquarium     | 3:1 |     |     |     |  | pigmented | 4:1         |
| are          | 3:1 | 4:1 |     |     |  | popular   | 3:1         |
| around       | 1:1 |     |     |     |  | refer     | 2:1         |
| as           | 2:1 |     |     |     |  | referred  | 2:1         |
| both         | 1:1 |     |     |     |  | requiring | 2:1         |
| bright       | 3:1 |     |     |     |  | salt      | 1:1 4:1     |
| coloration   | 3:1 | 4:1 |     |     |  | saltwater | 2:1         |
| derives      | 4:1 |     |     |     |  | species   | 1:1         |
| due          | 3:1 |     |     |     |  | term      | 2:1         |
| environments | 1:1 |     |     |     |  | the       | 1:1 2:1     |
| fish         | 1:2 | 2:3 | 3:2 | 4:2 |  | their     | 3:1         |
| fishkeepers  | 2:1 |     |     |     |  | this      | 4:1         |
| found        | 1:1 |     |     |     |  | those     | 2:1         |
| fresh        | 2:1 |     |     |     |  | to        | 2:2 3:1     |
| freshwater   | 1:1 | 4:1 |     |     |  | tropical  | 1:2 2:2 3:1 |
| from         | 4:1 |     |     |     |  | typically | 4:1         |
| generally    | 4:1 |     |     |     |  | use       | 2:1         |
| in           | 1:1 | 4:1 |     |     |  | water     | 1:1 2:1 4:1 |
| include      | 1:1 |     |     |     |  | while     | 4:1         |
| including    | 1:1 |     |     |     |  | with      | 2:1         |
| iridescence  | 4:1 |     |     |     |  | world     | 1:1         |
| marine       | 2:1 |     |     |     |  |           |             |
| often        | 2:1 | 3:1 |     |     |  |           |             |

(default)  
TF inv list (docs, counts)  
• no positions

inv. list ("freshwater")  
docid: count

docid  
TF = count (freshwater, docid)

**Inverted Index with Counts**

# Indexing Additional Data

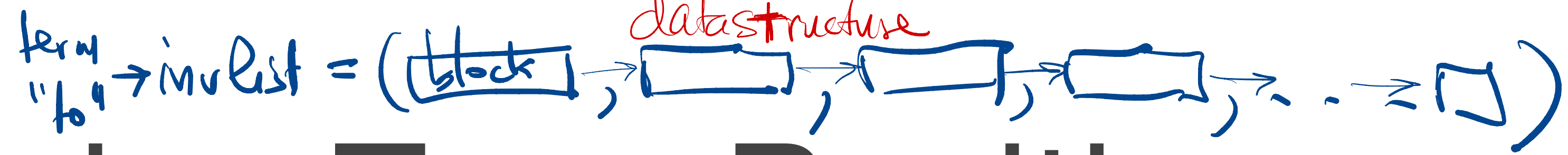
The information used to support all modern search features can grow quite complex.

Locations, dates, usernames, and other metadata are common search criteria, especially in search functions of web and mobile applications.

When these fields contain text, they are ultimately stored using the same inverted list structure.

Next, we'll see how to compress inverted lists to reduce storage needs and filesystem I/O.

# Indexing Term Positions

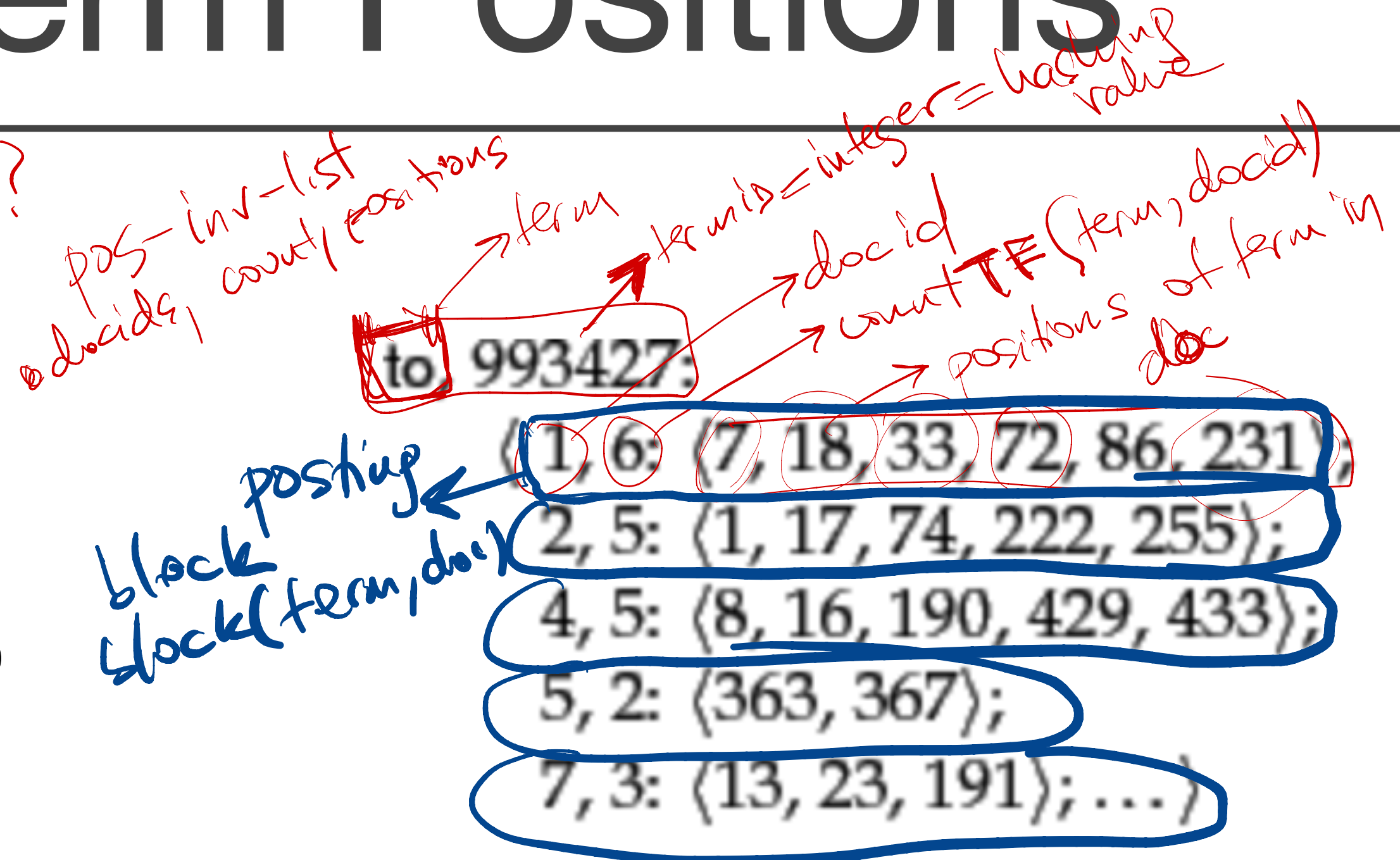


Many scoring functions assign higher scores to documents containing the query terms in closer proximity.

Some query languages allow users to specify proximity requirements, like “tropical NEAR fish.”

In the inverted lists to the right, the word “to” has a DF of 993,427. It is found in five documents; its TF in doc 1 is 6, and the list of positions is given.

*block output?*



be 178239:  
(1, 2: (17, 25);  
4, 5: (17, 191, 291, 430, 434);  
5, 3: (14, 19, 101); ...)

Postings with DF, TF, and Positions

# Proximity Searching

In proximity search, you search for documents where terms are sufficiently close to each other.

We process terms from least to most common in order to minimize the number of documents processed.

The algorithm shown here finds documents from two inverted lists where the terms are within  $k$  words of each other.

```
POSITIONALINTERSECT( $p_1, p_2, k$ )
1   $answer \leftarrow \langle \rangle$ 
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3  do if  $\text{docID}(p_1) = \text{docID}(p_2)$ 
4      then  $l \leftarrow \langle \rangle$ 
5           $pp_1 \leftarrow \text{positions}(p_1)$ 
6           $pp_2 \leftarrow \text{positions}(p_2)$ 
7          while  $pp_1 \neq \text{NIL}$ 
8          do while  $pp_2 \neq \text{NIL}$ 
9              do if  $|\text{pos}(pp_1) - \text{pos}(pp_2)| \leq k$ 
10                 then  $\text{ADD}(l, \text{pos}(pp_2))$ 
11                 else if  $\text{pos}(pp_2) > \text{pos}(pp_1)$ 
12                     then break
13                      $pp_2 \leftarrow \text{next}(pp_2)$ 
14                 while  $l \neq \langle \rangle$  and  $|l[0] - \text{pos}(pp_1)| > k$ 
15                     do  $\text{DELETE}(l[0])$ 
16                 for each  $ps \in l$ 
17                     do  $\text{ADD}(answer, \langle \text{docID}(p_1), \text{pos}(pp_1), ps \rangle)$ 
18                  $pp_1 \leftarrow \text{next}(pp_1)$ 
19                  $p_1 \leftarrow \text{next}(p_1)$ 
20                  $p_2 \leftarrow \text{next}(p_2)$ 
21             else if  $\text{docID}(p_1) < \text{docID}(p_2)$ 
22                 then  $p_1 \leftarrow \text{next}(p_1)$ 
23                 else  $p_2 \leftarrow \text{next}(p_2)$ 
24 return  $answer$ 
```

**Algorithm for Proximity Search**

# Indexing Scores

---

For some search applications, it's worth storing the document's matching score for a term in the posting list.

Postings may be sorted from largest to smallest score, in order to quickly find the most relevant documents. This is especially useful when you want to quickly find the approximate-best documents rather than the exact-best.

Indexing scores makes queries much faster, but gives less flexibility in updating your retrieval function. It is particularly efficient for single term queries.

For Machine Learning based retrieval, it's common to store per-term scores such as BM25 as features.

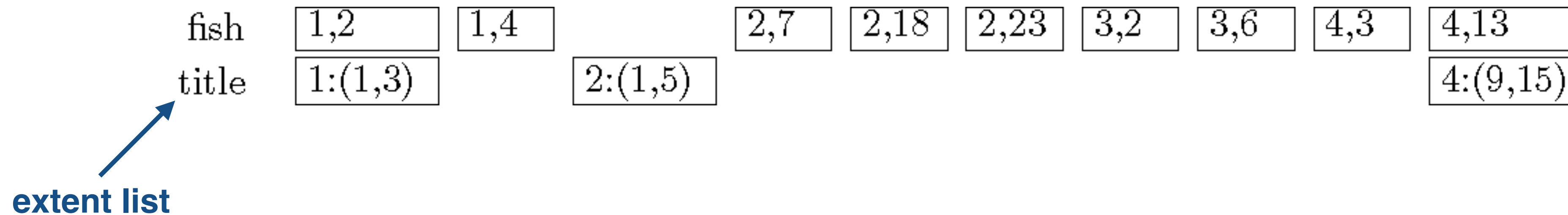


# Fields and Extents

Some indexes have distinct *fields* with their own inverted lists. For instance, an index of e-mails may contain fields for common e-mail headers (from, subject, date, ...).

Others store document regions such as the title or headers using *extent lists*.

Extent lists are contiguous regions of a document stored using term positions.



# Index Schemas

As the information stored in an inverted index grows more complex, it becomes useful to represent it using some form of schema.

However, we normally don't use strict SQL-type schemas, partly due to the cost of rebuilding a massive index. Instead, flexible formats such as `<key, value>` maps with field names arranged by convention are used.

Each text field in the schema typically gets its own inverted lists.

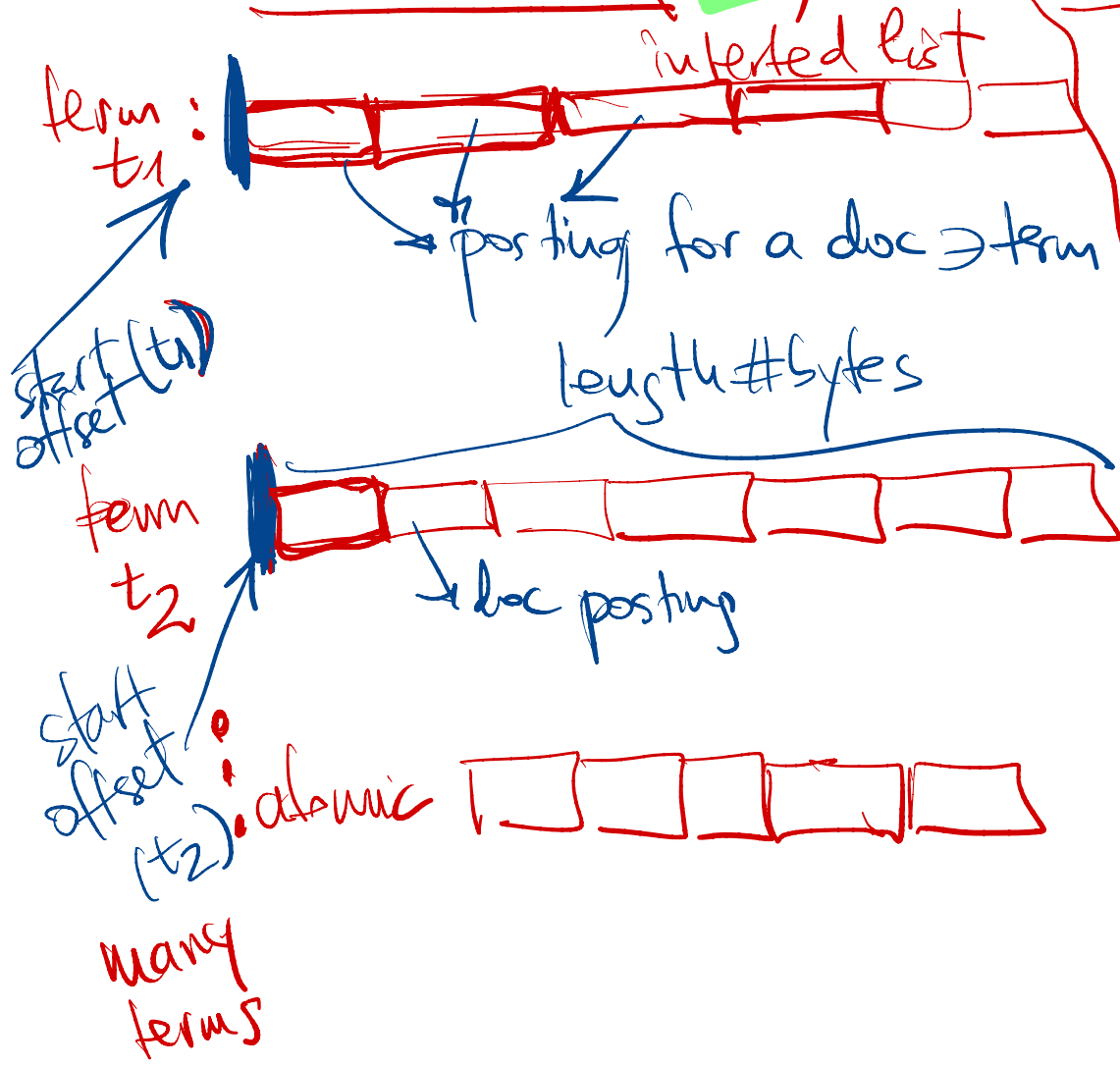
```
{
  "retweeted":false,
  "favorited":false,
  "created_at":"Thu Nov 17 19:02:46 +0000 2014",
  "in_reply_to_screen_name":null,
  "user":{
    "screen_name":"user01",
    "geo_enabled":true,
    "lang":"en",
    "time_zone":"Mountain Time (US & Canada)",
    "created_at":"Fri Sept 23 23:23:39 +0000 2010",
    "location":"Boise, ID",
  },
  "retweet_count":null,
  "id":12345678,
  "in_reply_to_user_id":null,
  "text":"just spent the day learning about lucene"
}
```

**Partial JSON Schema for Tweets**

(conceptually) inverted index one-to-one

inverted file (hdd)

catalog file (memory) → hash(t) → random access



$t_1$ : <sup>start</sup> offset (bytes), length offset

$t_2$ : start offset, length (bytes).

ex:

|        |         |          |
|--------|---------|----------|
| atomic | 12751   | 922      |
| :      | ↓ start | ↓ length |
| :      |         |          |
| :      |         |          |
| :      |         |          |

many terms

handle = open (inv. file)

- write (handle, start)
- read (length bytes)

① parse 1000 docs  $\Rightarrow$  inv file + catalog

1000 docs inv file 1, catalog 1

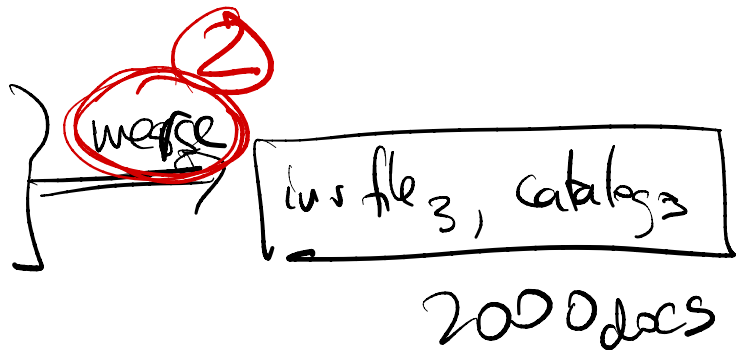
1000 docs inv file 2, catalog 2

1000 docs

⋮  
⋮  
⋮

85

1000 docs



1 parse docs  $\rightarrow$  inv file + catalog  
1000

for each doc  $d \in$  set of 1000

for each term  $t \in d$  // parse the document

ignore  $t$ ?  $\xrightarrow{\text{yes}}$  next

Block **hash(t)(d)** update

Block operation  
add current position to pos\_list  
update TF count

first key  
second key

done parsing

term  $t$

output inv-list(t)

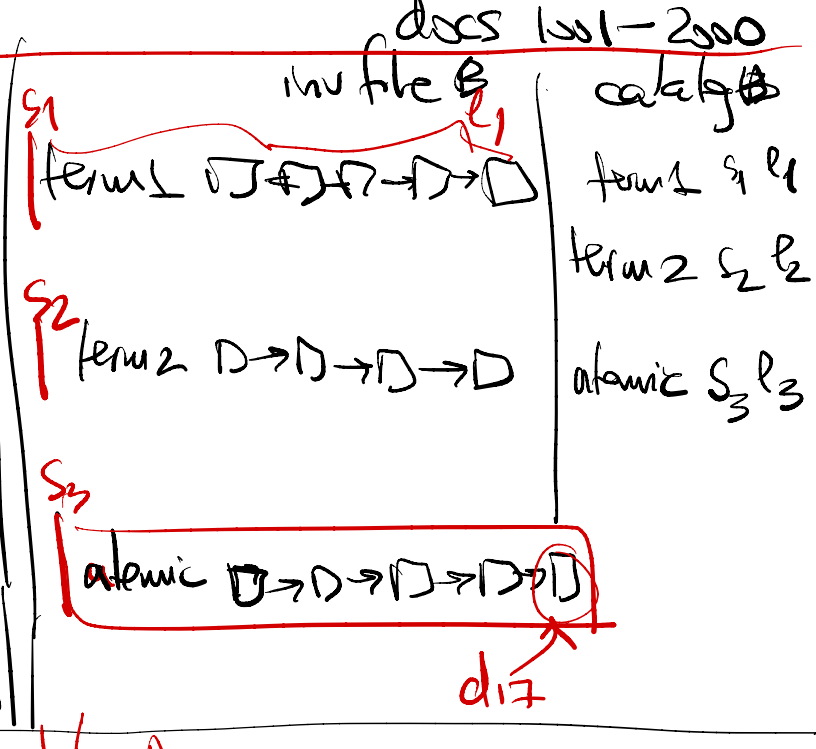
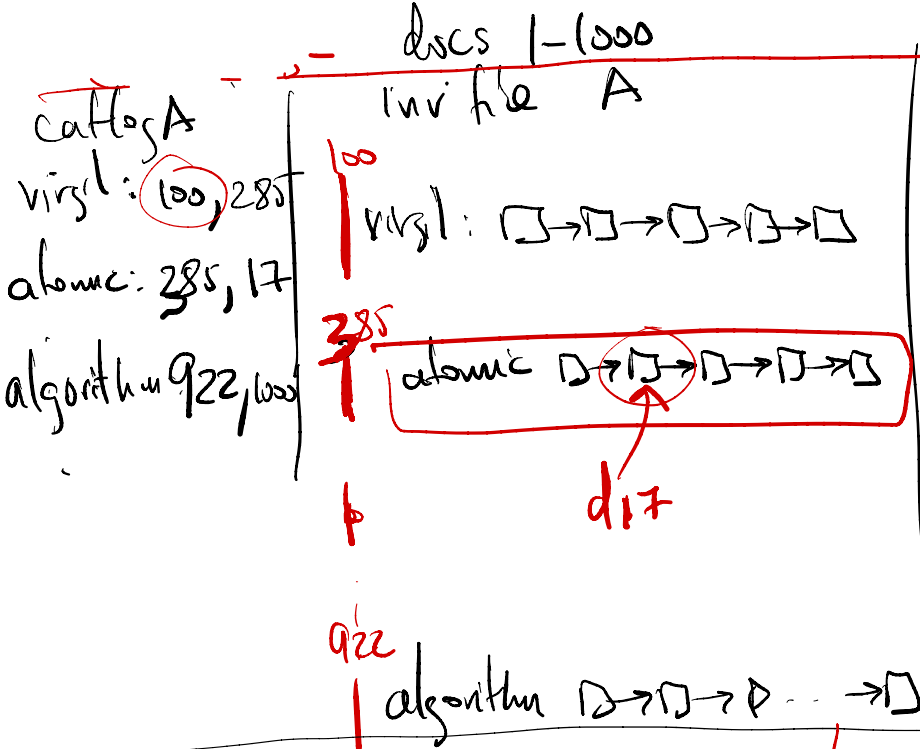
for each  $d \in$  Keys(hash(t))

write block **hash(t)(d)**

add inv file

t: [d1]  $\rightarrow$  [d2]  $\rightarrow$  [d3]  $\rightarrow$  [ ]  $\rightarrow$  [ ]

write catalog entry  
catalog for term  
t, start offset, length



MERGE → reap MERGE procedure from MERGESORT

by catalog A

for each term t ∈ catalog A

invlist\_A = read\_inv\_list(t, A)

if t ∈ catalog B then

invlist\_B = read\_inv\_list(t, B)



attribute  $s \in$   
 algorithm  $s \in$   
 term 1  $s \times$   
 term 2  $s \in$

$attr \ D \rightarrow D \rightarrow D$   
 merged

$inv\_list - C = \text{MERGE}(inv\_list A, inv\_list B)$   
 write  $inv\_list - C$  to new merge file  
 update catalog  $C$   $\rightarrow$  on the fly  
 else with  $inv\_list - A$  info new merge file  
 update catalog  $C$   $\rightarrow$  don't count on cat  $s \in A \cap B$

**MERGE** ( $inv\_list A, inv\_list B$ )  
 simply concat works thru  
assuming : no shared docs

for each term  $t \in$  catalog  $B$   
 if  $t \in$  catalog  $A \rightarrow$  next;  
 $inv\_list - B = read\_inv\_list(t, B)$   
 write  $inv\_list B$ , to new file merged  
 update catalog  $C$

not realistic in practice!

we might not control

$inv\_file A$   
 $inv\_file B$  } production  
 procedure similar to MergeSort

no assumption on docs indexed:

- concat too good.
- sort docs ids, apply MERGE

↳ close (file) ?

# Index Construction

We have just scratched the surface of the complexities of constructing and updating large-scale indexes. The most complex indexes are massive engineering projects that are constantly being improved.

An indexing algorithm needs to address hardware limitations (e.g., memory usage), OS limitations (the maximum number of files the filesystem can efficiently handle), and algorithmic concerns.

When considering whether your algorithm is sufficient, consider how it would perform on a document collection a few orders of magnitude larger than it was designed for.



# Basic Indexing

Given a collection of documents, how can we efficiently create an inverted index of its contents?

The basic steps are:

1. Tokenize each document, to convert it to a sequence of terms
2. Add doc to inverted list for each token

This is simple at small scale and in memory, but grows much more complex to do efficiently as the document collection and vocabulary grow.

## Basic In-Memory Indexer

```
def build_index(docs):
    index = {}
    docid = 0
    for doc in docs:
        docid += 1
        tokens = parse_doc(doc)
        tokens = set(tokens)
        for token in tokens:
            if token not in index:
                index[token] = []
            index[token].append(docid)
    return index
```

# Merging Lists

---

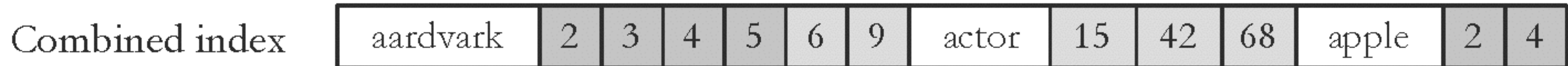
The basic indexing algorithm will fail as soon as you run out of memory.

To address this, we store a partial inverted list to disk when it grows too large to handle. We reset the in-memory index and start over. When we're finished, we merge all the partial indexes.

The partial indexes should be written in a manner that facilitates later merging. For instance, store the terms in some reasonable sorted order. This permits merging with a single linear pass through all partial lists.



# Merging Example



Admin 6/3/2020

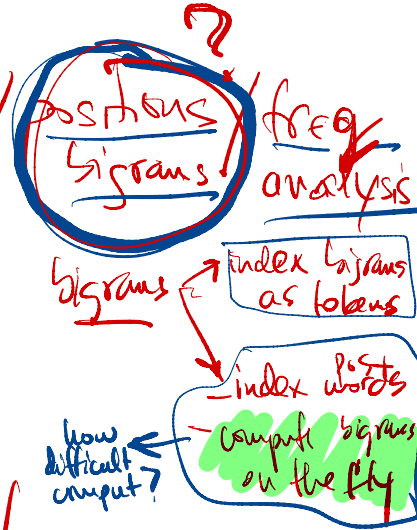
1 lecture behind  $\Rightarrow$  push HW2 deadline

next week  
Wed 6/10?

- Anjus demo ✓
  - = index construction :
- HW2
- encoding

encoding  
what to write

zip?  
compression  
how to write/  
compress



- positions n grams
- freq (natural freq) Heap's
- compression

Zipf's Laws

~~WRONG!~~ Hw2 naive non-merge?

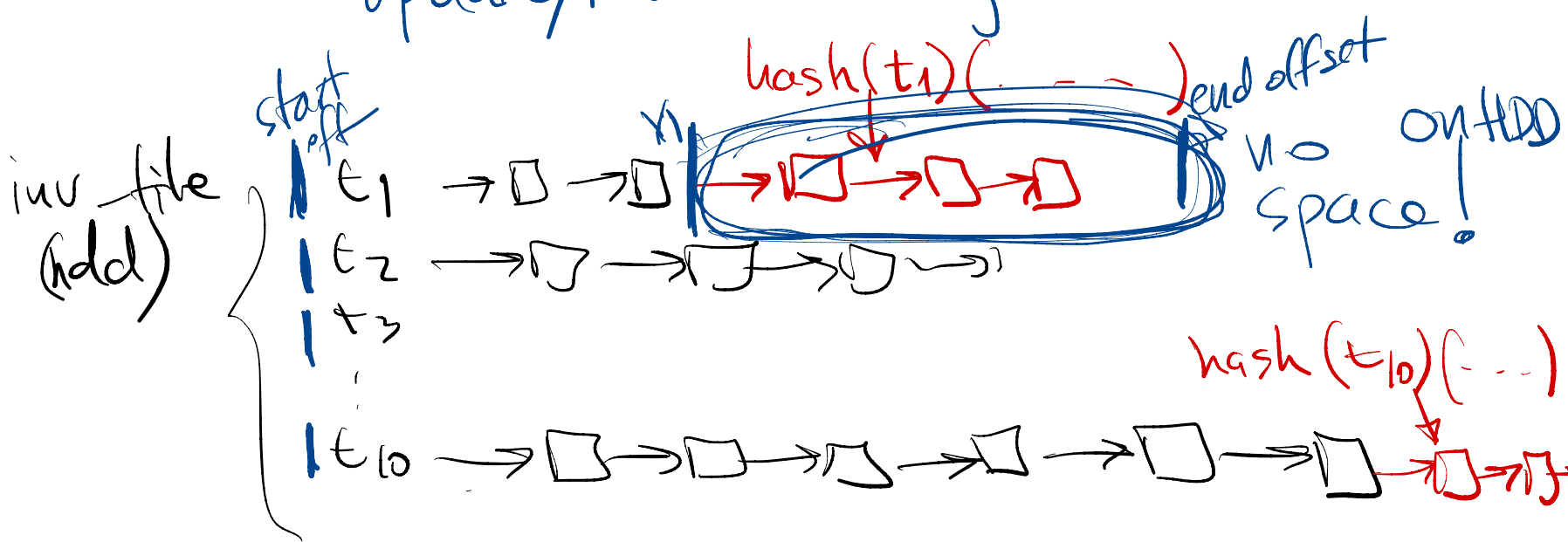
inv\_file = empty  
catalog = empty

For each batch of 1000 docs

hash = empty  
- parse, store data in  $hash(t)(d) = dir\_list\_block(t, d)$   
(same last time  $t \rightarrow \square \rightarrow \square \rightarrow \square \rightarrow \square \rightarrow \square$ )

- add  $hash(t)(d)$  to the existing inv file  
- update/rewrite catalog

85 batches



# Result Merging

---

An index can be updated from a new batch of documents by merging the posting lists from the new documents. However, this is inefficient for small updates.

Instead, we can run a search against both old and new indexes and merge the *result lists* at search time. Once enough changes have accumulated, we can merge the old and new indexes in a large batch.

In order to handle deleted documents, we also need to maintain a *delete list* of docids to ignore from the old index. At search time, we simply ignore postings from the old index for any docid in the delete list.

If a document is modified, we place its docid into the delete list and place the new version in the new index.

# Updating Indexes

---

If each term's inverted list is stored in a separate file, updating the index is straightforward: we simply merge the postings from the old and new index.

However, most filesystems can't handle very large numbers of files, so several inverted lists are generally stored together in larger files. This complicates merging, especially if the index is still being used for query processing.

There are ways to update live indexes efficiently, but it's often simpler to simply write a new index, then redirect queries to the new index and delete the old one.



# Compressing Indexes

The best any compression scheme can do depends on the entropy of the probability distribution over the data. More random data is less compressible.

Huffman Codes meet the entropy limit and can be built in linear time, so are a common choice. Other schemes can do better, generally by interpreting the input sequence differently (e.g. encoding sequences of characters as if they were a single input symbol – different distribution, different entropy limit).

# Index Size

---

Inverted lists often consume a large amount of space.

- e.g., 25-50% of the size of the raw documents for TREC collections with the Indri search engine
- much more than the raw documents if n-grams are indexed

Compressing indexes is important to conserve disk and/or RAM space. Inverted lists have to be decompressed to read them, but there are fast, lossless compression algorithms with good compression ratios.

frequency  $\rightarrow$  [efficiency]  $\rightarrow$  security

us good  
fixed length

coding theory pipeline

term value object

number → write/encode (bits)

# restricted variable length codes

chars encode

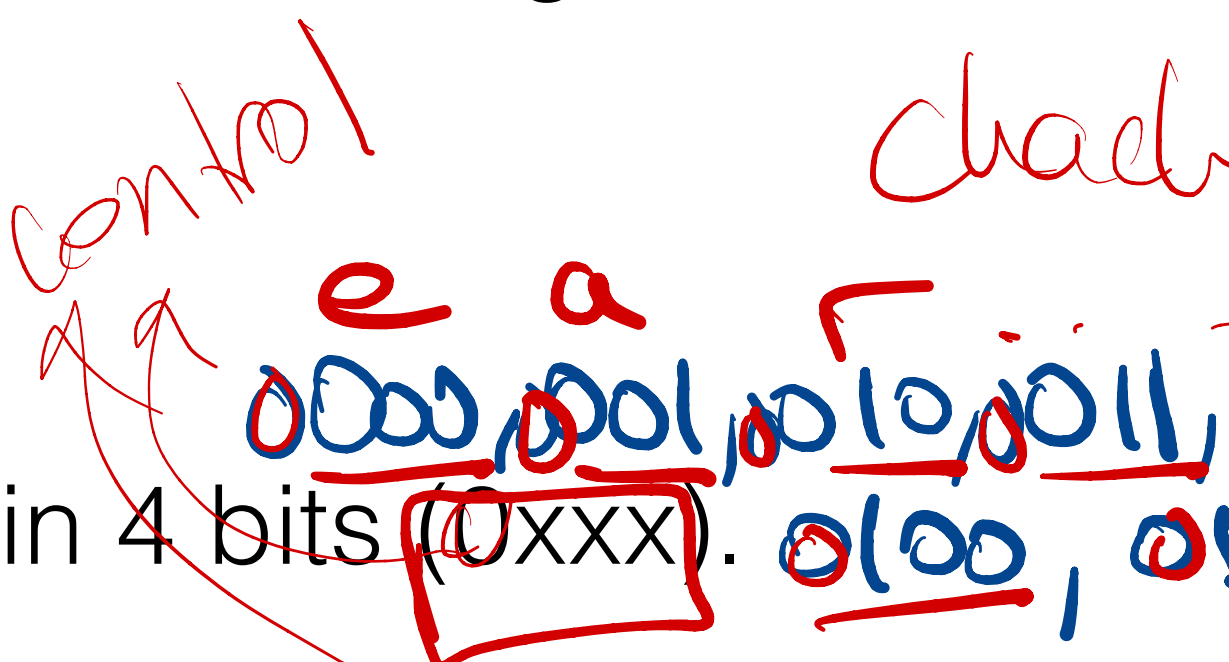
an extension of multibase encodings ("shift key") where different code lengths are used for each case. Only a few code lengths are chosen, to simplify encoding and decoding.

= ability to decode.

• Use first bit to indicate case.

2 3

• 8 most frequent characters fit in 4 bits (0xxx).



char  $\subseteq$  code (theory) principle  
high freq(x)  $\Rightarrow$  short encode (x)

2 7

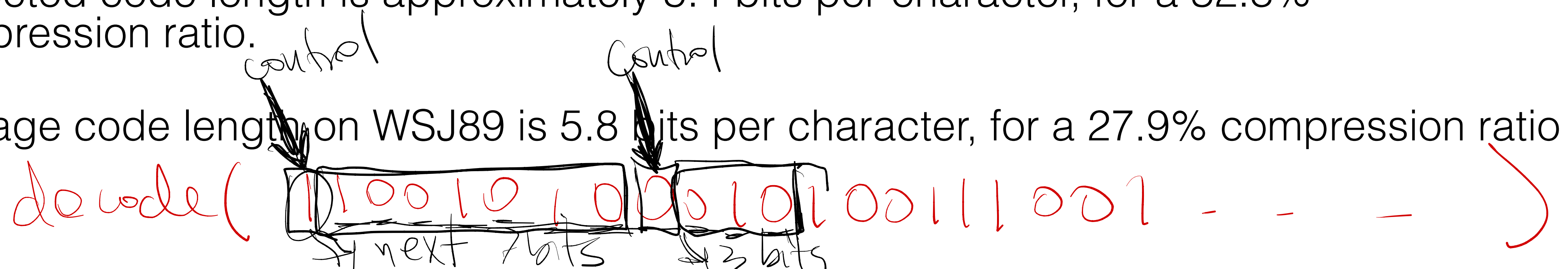
• 128 less frequent characters fit in 8 bits (1xxxxxxx)

3 bits for value.  
low freq(x)  $\Rightarrow$  long encode (x)  
7 bits for value

• In English, 7 most frequent characters are 65% of occurrences

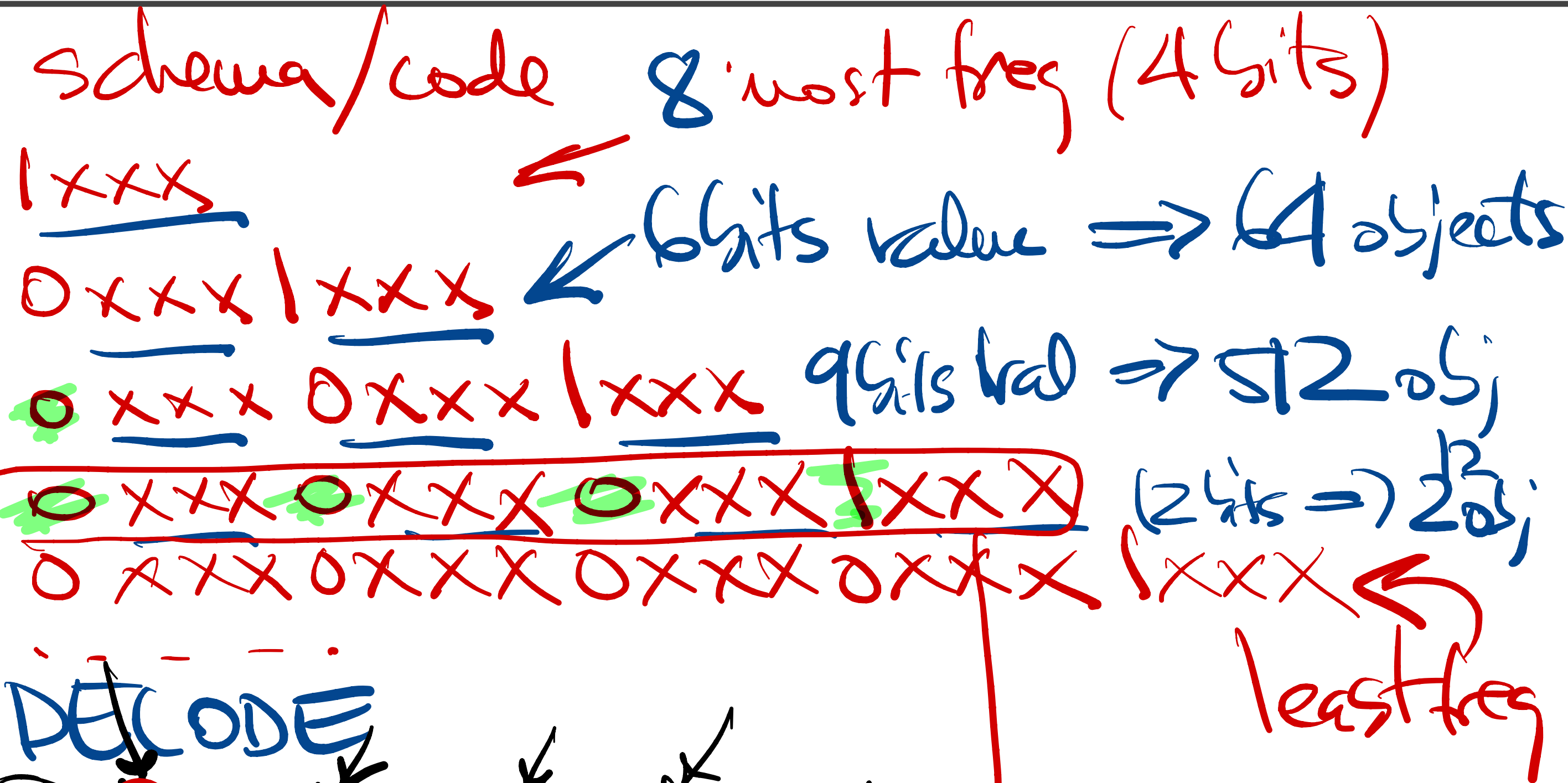
• Expected code length is approximately 5.4 bits per character, for a 32.8% compression ratio.

• average code length on WSJ89 is 5.8 bits per character, for a 27.9% compression ratio

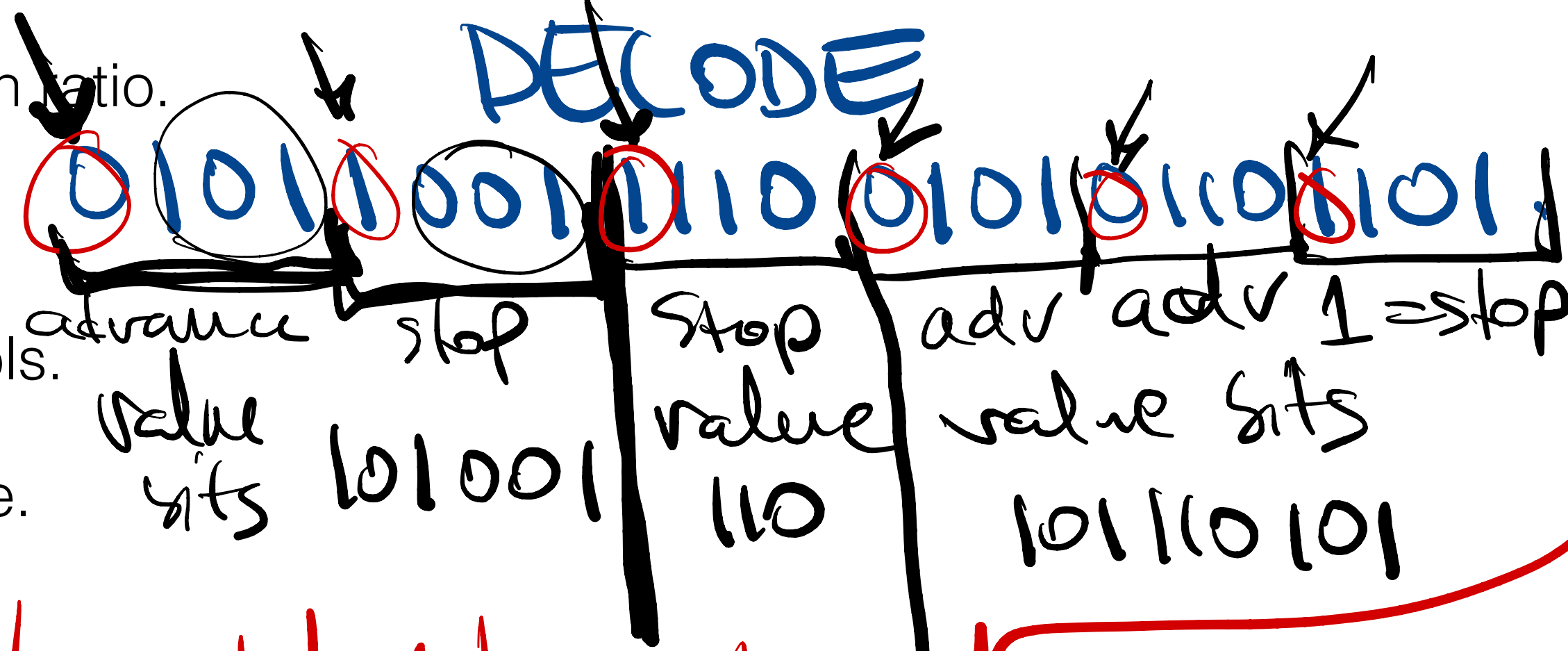


# restricted variable length codes: more symbols

- Use more than 2 cases.
- $1xxx$  for  $2^3 = 8$  most frequent symbols, and
- $0xxx1xxx$  for next  $2^6 = 64$  symbols, and
- $0xxx0xxx1xxx$  for next  $2^9 = 512$  symbols, and
- ...
- average code length on WSJ89 is 6.2 bits per symbol, for a 23.0% compression ratio.



- Pro: Variable number of symbols.
- Con: Only 72 symbols in 1 byte.



UBYTE: store all control bits first  $10001xxx\ xxx\ xxx\ xxx$

control

value

# restricted variable length codes : numeric data

---

- 1xxxxxxx for  $2^7 = 128$  most frequent symbols
- 0xxxxxxx1xxxxxxx for next  $2^{14} = 16,384$  symbols
- ...
- average code length on WSJ89 is 8.0 bits per symbol, for a 0.0% compression ratio (!!).
- Pro: Can be used for integer data
  - Examples: word frequencies, inverted lists

# restricted variable –length codes : word based encoding

---

- Restricted Variable-Length Codes can be used on words (as opposed to symbols)
- build a dictionary, sorted by word frequency, most frequent words first
- Represent each word as an offset/index into the dictionary
  
- Pro: a vocabulary of 20,000-50,000 words with a Zipf distribution requires 12-13 bits per word
  - compared with a 10-11 bits for completely variable length
- Con: The decoding dictionary is large, compared with other methods.

# restricted variable-length codes: summary

---

- Four methods presented. all are
  - simple
  - very effective when their assumptions are correct
- No assumptions about language or language models
- all require an unspecified mapping from symbols to numbers (a dictionary)
- all but the basic method can handle any size dictionary

next time : Zip (Lempel-Ziv) ; Huffman better than RVL

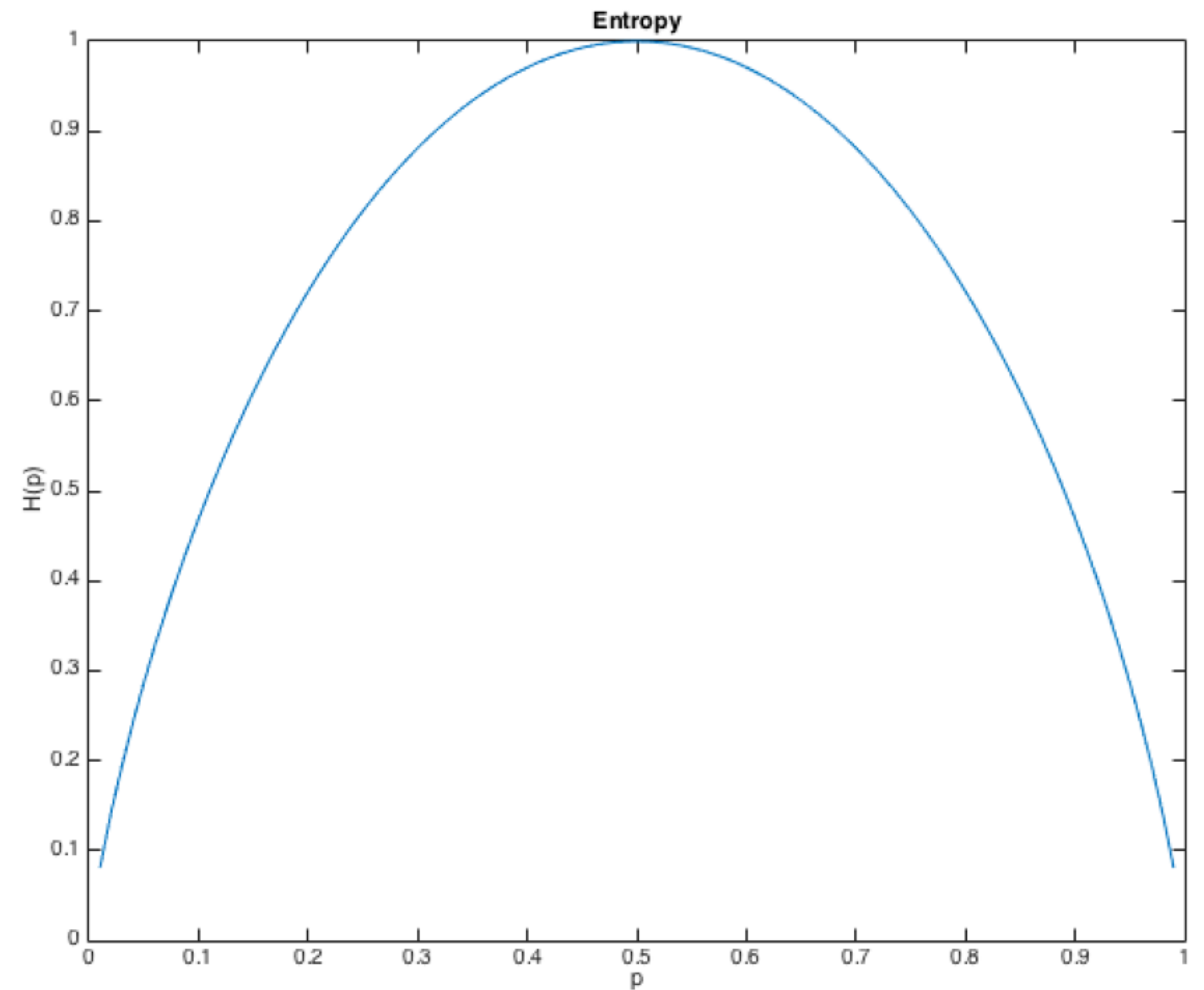
# Entropy and Compressibility

The **entropy** of a probability distribution is a measure of its randomness.

$$H(p) = - \sum_i p_i \log p_i$$

The more random a sequence of data is, the less predictable and less compressible it is.

The entropy of the probability distribution of a data sequence provides a bound on the best possible compression ratio.



Entropy of a Binomial Distribution



# Huffman Codes

In an ideal encoding scheme, a symbol with probability  $p_i$  of occurring will be assigned a code which takes  $\log(p_i)$  bits.

The more probable a symbol is to occur, the smaller its code should be. By this view, UTF-32 assumes a uniform distribution over all unicode symbols; UTF-8 assumes ASCII characters are more common.

**Huffman Codes** achieve the best possible compression ratio when the distribution is known and when no code can stand for multiple symbols.

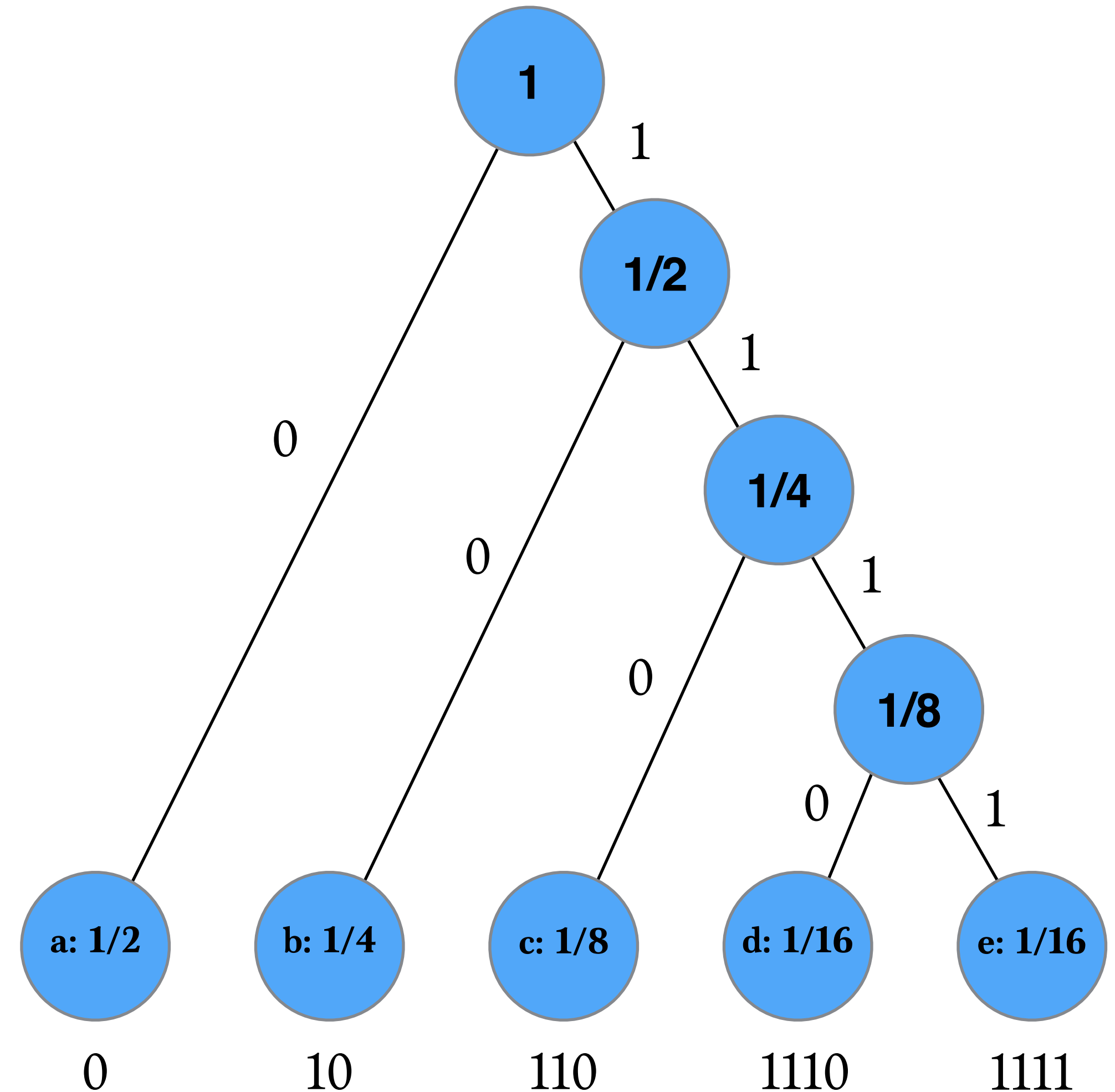
| Symbol | p    | Code | $\mathbb{E}[length]$ |
|--------|------|------|----------------------|
| a      | 1/2  | 0    | 0.5                  |
| b      | 1/4  | 10   | 0.5                  |
| c      | 1/8  | 110  | 0.375                |
| d      | 1/16 | 1110 | 0.25                 |
| e      | 1/16 | 1111 | 0.25                 |

**Plaintext:** aedbbaae (64 bits in UTF-8)  
**Ciphertext:** 0111111101010001111

# Building Huffman Codes

Huffman Codes are built using a binary tree which always joins the least probable remaining nodes.

1. Create a leaf node for each symbol, weighted by its probability.
2. Iteratively join the two least probable nodes without a parent by creating a parent whose weight is the sum of the children's weights.
3. Assign 0 and 1 to the edges from each parent. The code for a leaf is the sequence of edges on the path from the root.



# Can We Do Better?

---

Huffman codes achieve the theoretical limit for compressibility, assuming that the size of the code table is negligible and that each input symbol must correspond to exactly one output symbol.

Other codes, such as Lempel-Ziv encoding, allow variable-length sequences of input symbols to correspond to particular output symbols and do not require transferring an explicit code table.

Compression schemes such as gzip are based on Lempel-Ziv encoding. However, for encoding inverted lists it can be beneficial to have a 1:1 correspondence between code words and plaintext characters.

# Lempel-Ziv

---

- an adaptive dictionary approach to variable length coding.
- Use the text already encountered to build the dictionary.
- If text follows Zipf's laws, a good dictionary is built.
- No need to store dictionary; encoder and decoder each know how to build it on the fly.
- Some variants: LZ77, Gzip, LZ78, LZW, Unix **compress**
- Variants differ on:
  - how dictionary is built,
  - how pointers are represented (encoded), and
  - limitations on what pointers can refer to.

# Lempel Ziv: encoding

---

- 0010111010010111011011

# Lempel Ziv: encoding

---

- 0010111010010111011011
- break into known prefixes
- 0|01 |011|1 |010|0101|11|0110|11

# Lempel Ziv: encoding

---

- 0010111010010111011011
- break into known prefixes
- 0|01 |011|1 |010|0101|11|0110|11
- encode references as pointers
- 0|1,1|1,1 |0,1|3,0 |1,1 |3,1|5,0 |2,?

# Lempel Ziv: encoding

---

- 0010111010010111011011
- break into known prefixes
- 0|01 |011|1 |010|0101|11|0110|11
- encode references as pointers
- 0|1,1|1,1 |0,1|3,0 |1,1 |3,1|5,0 |2,?
- encode the pointers with  $\log(?)$ bits
- 0|1,1|01,1 |00,1|011,0 |001,1 |011,1|101,0 |0010,?



# Lempel Ziv: encoding

---

- 0010111010010111011011
- break into known prefixes: 0|01 |011|1 |010|0101|11|0110|11
- encode references as pointers : 0|1,1|1,1 |0,1|3,0 |1,1 |3,1|5,0 |2,?
- encode the pointers with  $\log(?)$ bits :  
0|1,1|01,1 |00,1|011,0 |001,1 |011,1|101,0 |0010,?
- final string : 01101100101100011011110100010

# Lempel Ziv: decoding

---

- 01101100101100011011110100010

# Lempel Ziv: decoding

---

- 01101100101100011011110100010
- decode the pointers with  $\log(?)$ bits
- 0|1,1|01,1 |00,1|011,0 |001,1 |011,1|101,0 |0010,?

# Lempel Ziv: decoding

---

- 01101100101100011011110100010
- decode the pointers with  $\log(?)$ bits
- 0|1,1|01,1 |00,1|011,0 |001,1 |011,1|101,0 |0010,?
- encode references as pointers
- 0|1,1|1,1 |0,1|3,0 |1,1 |3,1|5,0 |2,?

# Lempel Ziv: decoding

---

- 01101100101100011011110100010
- decode the pointers with  $\log(?)$ bits
- 0|1,1|01,1 |00,1|011,0 |001,1 |011,1|101,0 |0010,?
- encode references as pointers
- 0|1,1|1,1 |0,1|3,0 |1,1 |3,1|5,0 |2,?
- decode references
- 0|01 |011|1 |010|0101|11|0110|11

# Lempel Ziv: decoding

---

- 01101100101100011011110100010
- decode the pointers with  $\log(?)$  bits : 0|1,1|01,1 |00,1|011,0 |001,1 |011,1|101,0 |0010,?
- encode references as pointers : 0|1,1|1,1 |0,1|3,0 |1,1 |3,1|5,0 |2,?
- decode references : 0|01 |011|1 |010|0101|11|0110|11
- original string : 0010111010010111011011

# Lempel Ziv optimality

---

- LempelZiv compression rate approaches (asymptotic) entropy
- When the strings are generated by an ergodic source [CoverThomas91].
- easier proof : for i.i.d sources
  - that is not a good model for English

# LempelZiv optimality

## -i.i.d source

---

- let  $x = \alpha_1\alpha_2\dots\alpha_n$  a sequence of length  $n$  generated by a iid source and  $Q(x) =$  the probability to see such a sequence

- say LempelZiv breaks into  $c$  phrases  $x = y_1y_2\dots y_c$  and call  $c_l = \#$  of phrases of length  $l$  then  $-\log Q(x) \geq \sum_l c_l \log c_l$

(proof)  $\sum_{|y_i|=l} Q(y_i) < 1$  so  $\prod_{|y_i|=l} Q(y_i) < (\frac{1}{c_l})^{c_l}$

- if  $p_i$  is the source probab for  $\alpha_i$  then by law of large numbers  $x$  will have roughly  $np_i$  occurrences of  $\alpha_i$  and then

$$\log Q(x) = -\log \prod_i p_i^{np_i} \approx n \sum p_i \log p_i = nH_{source}$$

- note that  $\sum_l c_l \log c_l$  is roughly the LempelZiv encoding length so th inequality reads  $nH \geq \approx LZencoding$  which is to say  $H \approx \geq LZrate$ .



# Bit-aligned Codes

Bit-aligned codes allow us to minimize the storage used to encode integers.

We can use just a few bits for small integers, and still represent arbitrarily large numbers.

Inverted lists can also be made more compressible by delta-encoding their contents.

Next, we'll see how to encode integers using a variable byte code, which is more convenient for processing.

# Compressing Inverted Lists

An inverted list is generally represented as multiple sequences of integers.

- Term and document IDs are used instead of the literal term or document URL/path/name.
- TF, DF, term position lists and other data in the inverted lists are often integers.

We'd like to efficiently encode this integer data to help minimize disk and memory usage. But how?

**to, 993427:**

```
⟨ 1, 6: ⟨7, 18, 33, 72, 86, 231⟩;  
  2, 5: ⟨1, 17, 74, 222, 255⟩;  
  4, 5: ⟨8, 16, 190, 429, 433⟩;  
  5, 2: ⟨363, 367⟩;  
  7, 3: ⟨13, 23, 191⟩; ... ⟩
```

**be, 178239:**

```
⟨ 1, 2: ⟨17, 25⟩;  
  4, 5: ⟨17, 191, 291, 430, 434⟩;  
  5, 3: ⟨14, 19, 101⟩; ... ⟩
```

**Postings with DF, TF, and Positions**

# Unary

The encodings used by processors for integers (e.g., two's complement) use a fixed-width encoding with fixed upper bounds. Any number takes 32 (say) bits, with no ability to encode larger numbers.

Both properties are bad for inverted lists. Smaller numbers tend to be much more common, and should take less space. But very large numbers can happen – consider term positions in very large files, or document IDs in a large web collection.

What if we used a **unary** encoding? This encodes  $k$  by  $k$  **1**s, followed by a **0**.

| decimal | binary   | unary          |
|---------|----------|----------------|
| 0       | 00000000 | 0              |
| 1       | 00000001 | 10             |
| 7       | 00000111 | 11111110       |
| 13      | 00001101 | 11111111111110 |

# Elias- $\gamma$ Codes

Unary is efficient for small numbers, but very inefficient for large numbers. There are better ways to get a variable bit length.

With Elias- $\gamma$  codes, we use unary to encode the bit length and then store the number in binary.

To encode a number  $k$ , compute:

$$\left\{ \begin{array}{l} k_d = \lfloor \log_2 k \rfloor \\ k_r = k - 2^{\lfloor \log_2 k \rfloor} \end{array} \right. \quad \begin{array}{l} \text{save 2 logs} \\ \text{diff to power of 2} \end{array}$$

$k=15$   
 $k_d = \lfloor \log_2 15 \rfloor = 3$   
 $k_r = 15 - 2^3 = 7$

| Decimal | $k_d$ | $k_r$ | 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<br>111111111 |

binary part( $k_r$ ) uses  $k_d$  bits (enough?)

bits  
decode

control  
binary-how many bits

3 unary

9 binary

# Elias- $\delta$ Codes

$\underbrace{1111}_0 \underbrace{1101}_1 \mid \begin{matrix} k_d = 4 \\ k_r = 13 \end{matrix}$   
 value =  $2^{k_d} + k_r = 29$

Elias- $\gamma$  codes take  $2\lfloor \log_2 k \rfloor + 1$  bits. We can do better, especially for large numbers.

Elias- $\delta$  codes encode  $k_d$  using an Elias- $\gamma$  code, and take approximately  $2\log_2 \log_2 k + \log_2 k$  bits.

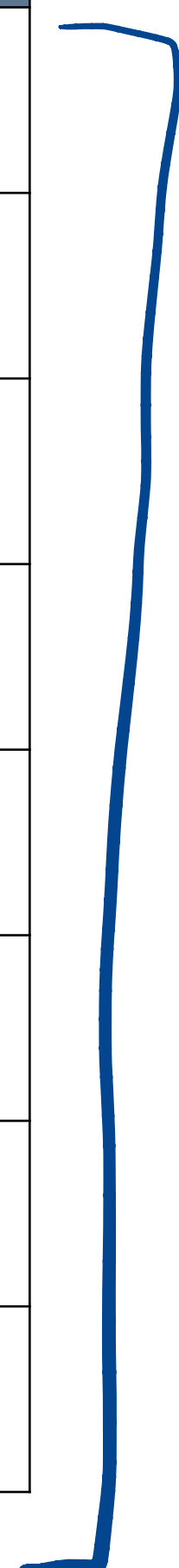
We split  $k_d$  into:

$$k_{dd} = \lfloor \log_2 k_d \rfloor$$

$$k_{dr} = k_d - 2^{\lfloor \log_2 k_d \rfloor}$$

*double encode*

| Decimal | $k$ | $k$ | $k$ | $k$ | Code                  |
|---------|-----|-----|-----|-----|-----------------------|
| 1       | 0   | 0   | 0   | 0   | 0                     |
| 2       | 1   | 1   | 0   | 0   | 10 0 0                |
| 3       | 1   | 1   | 0   | 1   | 10 0 1                |
| 6       | 2   | 1   | 1   | 2   | 10 1 10               |
| 15      | 3   | 2   | 0   | 7   | 110 00 111            |
| 16      | 4   | 2   | 1   | 0   | 110 01 0000           |
| 255     | 7   | 3   | 0   | 127 | 1110 000<br>1111111   |
| 1023    | 9   | 3   | 2   | 511 | 1110 010<br>111111111 |



# Python Implementation

```
1 import math
2
3 def unary_encode(n):
4     return "1" * n + "0"
5
6 def binary_encode(n, width):
7     r = ""
8     for i in range(0, width):
9         if ((1<<i) & n) > 0:
10            r = "1" + r
11        else:
12            r = "0" + r
13    return r
14
15 def gamma_encode(n):
16     logn = int(math.log(n,2))
17     return unary_encode(logn) + " " + binary_encode(n, logn)
18
19 def delta_encode(n):
20     logn = int(math.log(n,2))
21     if n == 1:
22         return "0"
23     else:
24         loglog = int(math.log(logn+1, 2))
25         residual = logn+1 - int(math.pow(2, loglog))
26         return (unary_encode(loglog) + " "
27                + binary_encode(residual, loglog) + " "
28                + binary_encode(n, logn))
```

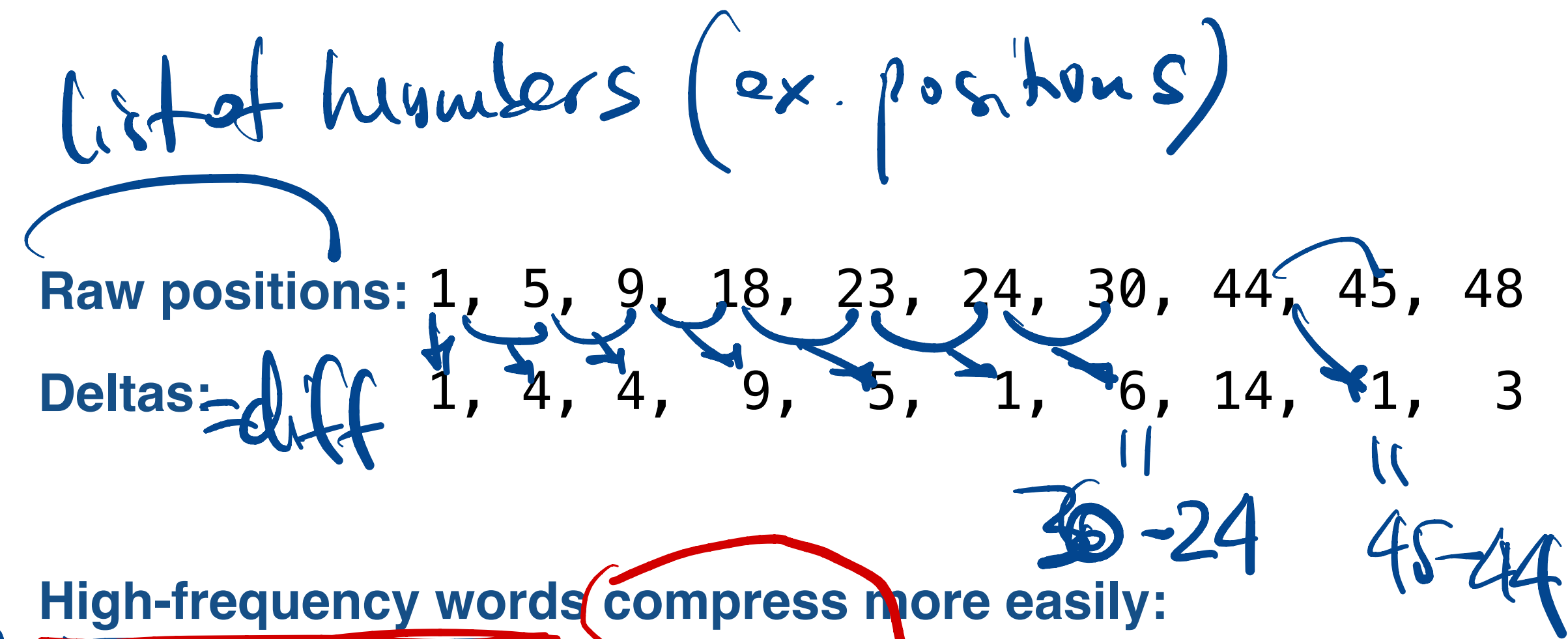
```
30 if __name__ == '__main__':
31     for n in [1, 2, 3, 6, 15, 16, 255, 1023]:
32         logn = int(math.log(n,2))
33         loglogn = int(math.log(logn+1,2))
34         print n, "d_r", logn
35         print n, "d_dd", loglogn
36         print n, "d_dr", logn + 1 - int(math.pow(2, loglogn))
37         print n, "delta", delta_encode(n)
```

# Delta Encoding

We now have an efficient variable bit length integer encoding scheme which uses just a few bits for small numbers, and can handle arbitrarily large numbers with ease.

To further reduce the index size, we want to ensure that docids, positions, etc. in our lists are small (for smaller encodings) and repetitive (for better compression).

We can do this by sorting the lists and encoding the difference, or delta, between the current number and the last.



- High-frequency words compress more easily:  
1, 1, 2, 1, 5, 1, 4, 1, 1, 3, ...  
Small Deltas  $\Rightarrow$  repeating deltas/patterns
- Low-frequency words have larger deltas:  
109, 3766, 453, 1867, 992, ...  
larger deltas  $\Rightarrow$  non repeating

# Byte-Aligned Codes

In production systems, inverted lists are stored using byte-aligned codes for delta-encoded integer sequences.

Careful engineering of encoding schemes can help tune this process to minimize processing while reading the inverted lists. This is essential for getting good performance in high-volume commercial systems.

Next, we'll look at how to produce an index from a document collection.



# Byte-Aligned Codes

---

We've looked at ways to encode integers with bit-aligned codes. These are very compact, but somewhat inconvenient.

Processors and most I/O routines and hardware are byte-aligned, so it's more convenient to use byte-aligned integer encodings.

One of the commonly-used encodings is called **vbyte**. This encoding, like UTF-8, simply uses the most significant bit to encode whether the number continues to the next byte.

# Vbyte

| $k$ | Bytes Used |
|-----|------------|
| $k$ | 1          |
| 2   | 2          |
| 2   | 3          |
| 2   | 4          |

| $k$   | Binary                        | 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    |

# Java Implementation

---

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

```
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);
    }
}
```

# Bringing It Together

---

Let's see how to put together a compressed inverted list with delta encoding. We start with the raw inverted list: a sequence of tuples containing (`docid`, `tf`, [`pos1`, `pos2`, ...]).

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

We delta-encode the docid and position sequences independently.

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

Finally, we encode the integers using `vbyte`.

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

# Alternative Codes

---

Although vbyte is often adequate, we can do better for high-performance decoding.

Vbyte requires a conditional branch at every byte and a lot of bit shifting.

Google's Group VarInt encoding achieves much better decoding performance by storing a two bit continuation sequence for each of the next 4-16 bytes.

|                 |          |          |          |          |          |          |          |          |
|-----------------|----------|----------|----------|----------|----------|----------|----------|----------|
| <b>Decimal:</b> | 1        | 15       | 511      | 131071   |          |          |          |          |
| <b>Encoded:</b> | 00000110 | 00000001 | 00001111 | 11111111 | 00000001 | 11111111 | 11111111 | 00000001 |