Search Engines--Information Retrieval in Practice

tanshuqiu

Email: tsq@cqut.edu.cn

1

Ranking with Indexes

First : Overview

Second : Abstract Model of Ranking

Third : Inverted Indexes

Fourth : Compression

2

Ranking with Indexes

Fifth : Auxiliary Structures

Sixth : Index Construction

Seventh : Query Processing

Overview

Contents

- ✓Data structures
- ✓Inverted index
- ✓The relationship between components

Data structures

If you want to store a list of items, linked lists and arrays are good choices. If you want to quickly find an item based on an attribute, a hash table is a better choice. More complicated tasks require more complicated structures, such as B-trees or priority queues.

Inverted index

Text search is very different from traditional computing tasks, so it calls for its own kind of data structure, the inverted index. the specific kind of data structure used depends on the ranking function. since the ranking functions that rank documents well have a similar form, the most useful kinds of inverted indexes are found in nearly every search engine.

Inverted index

Efficient query processing is a particularly important problem in web search, as it has reached a scale that would have been hard to imagine just 10 years ago. People all over the world type in over half a billion queries every day, searching indexes containing billions of web pages. Inverted indexes are at the core of all modern web search engines.

The relationship between components

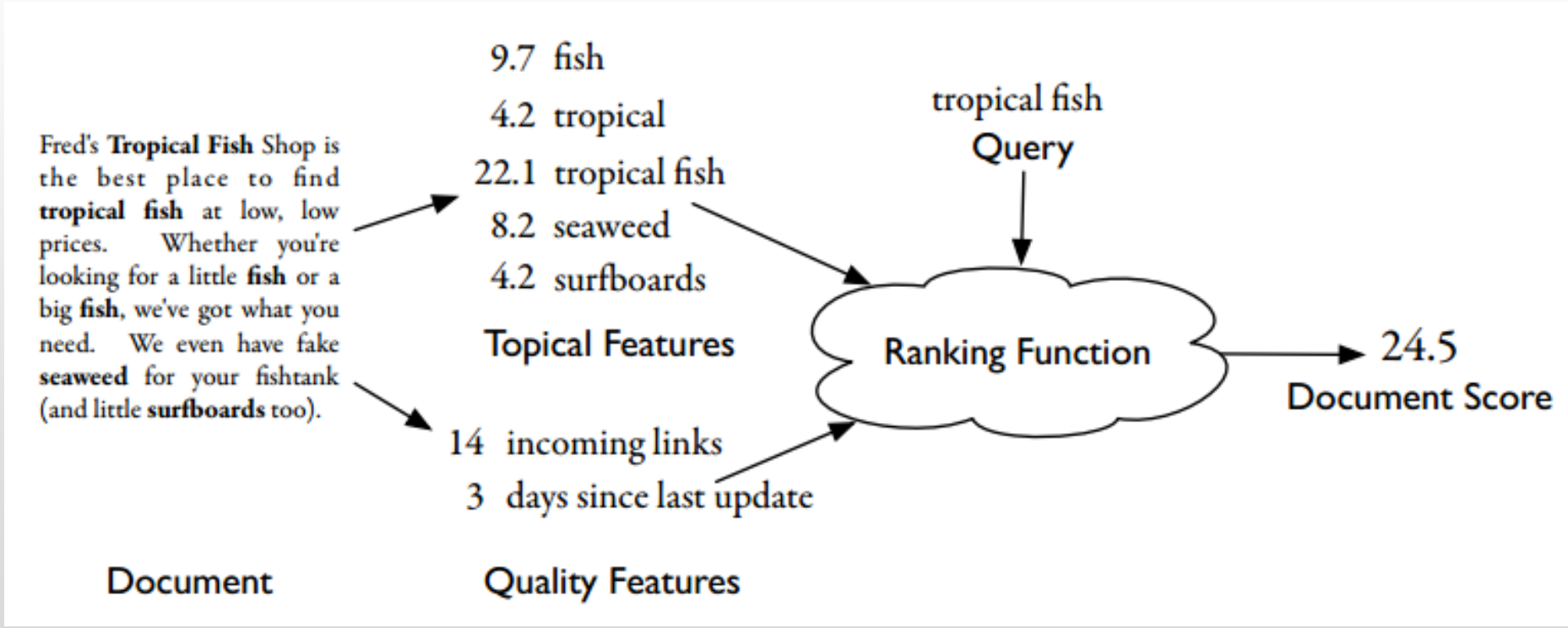
There are strong dependencies between the separate components of a search engine. The query processing algorithm depends on the retrieval model, and dictates the contents of the index.

Abstract Model of Ranking

Contents

- ✓The abstract model of ranking
- ✓A more concrete model of ranking

The abstract model of ranking



The abstract model of ranking

The components of the abstract model of ranking:

- documents
- features
- queries
- the retrieval function
- document scores

The abstract model of ranking

Documents are written in natural human languages, which are difficult for computers to analyze directly. So this documents need to transform into index terms or document features. a document feature is some attribute of the document we can express numerically.

The abstract model of ranking

There are two kinds of **features:**

topical features, which estimate the degree to which the document is about a particular subject.

quality features. One feature is the number of web pages that link to this document, and another is the number of days since this page was last updated.

Each of these feature values is generated using a feature function, which is just a mathematical expression that generates numbers from document text.

The abstract model of ranking

The **ranking function** takes data from document features combined with the query and produces a score.

The final output of the ranking function is a **score**, If a document gets a high score, this means that the system thinks that document is a good match for the query, whereas lower numbers mean that the system thinks the document is a poor match for the query.

To build a ranked list of results, the documents are sorted by score so that the highest-scoring documents come first

A more concrete model of ranking

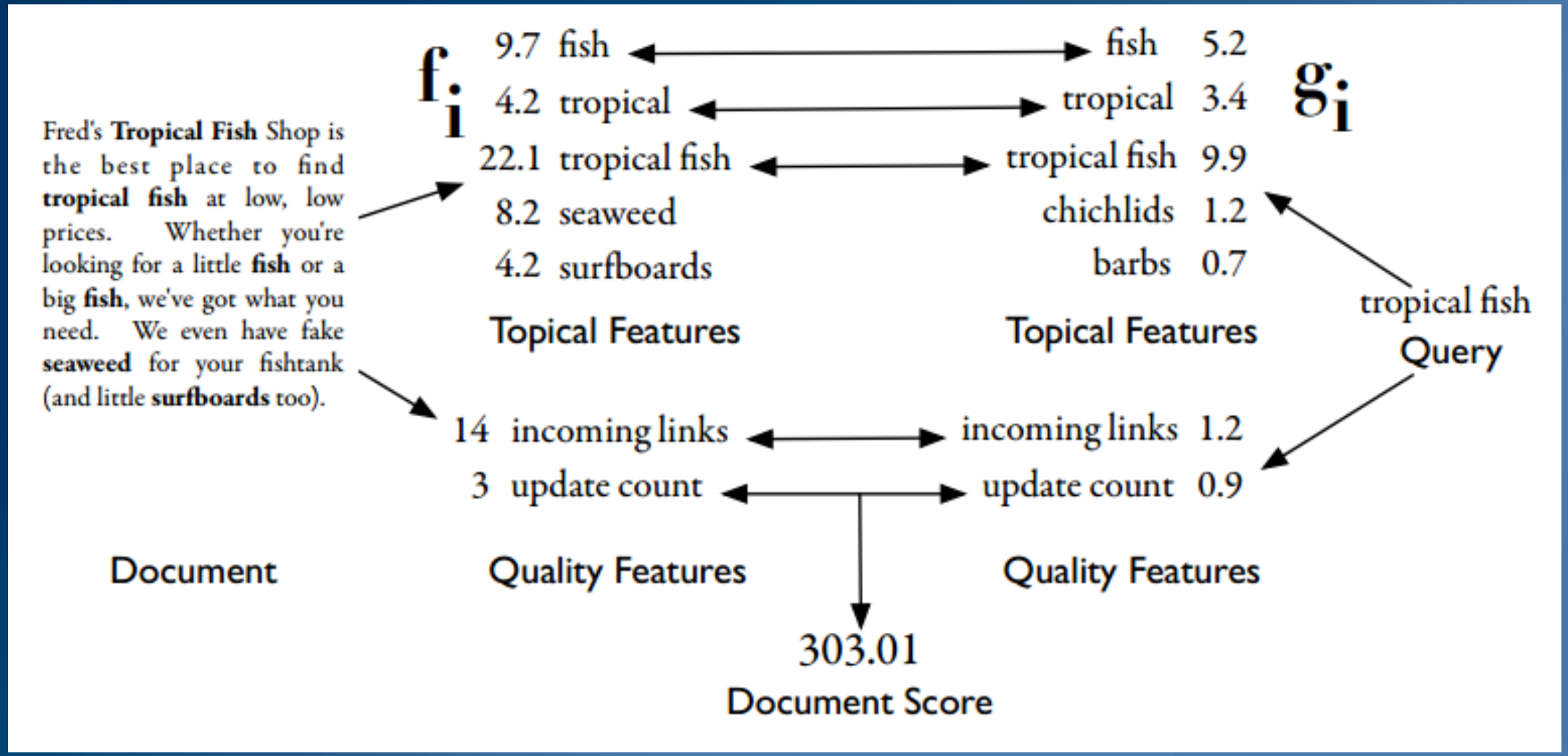
Query evaluation techniques: we assume that the ranking function R takes the following form.

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

Here, f_i is some feature function that extracts a number from the document text. g_i is a similar feature function that extracts a value from the query.

A more concrete model of ranking

These two functions f_i and g_i form a pair of feature functions. Each pair of functions is multiplied together, and the results from all pairs are added to create a final document score.



A more concrete model of ranking

A more concrete model of ranking

In the abstract model of ranking, various features are extracted from the document. These correspond to the $f_i(D)$ functions in the equation just shown. We could easily name these $f_{\text{tropical}}(D)$ or $f_{\text{fish}}(D)$; these values will be larger for documents that contain the words "tropical" or "fish" more often or more prominently.

A more concrete model of ranking

The document has some features that are not topical. For this example document, we see that the search engine notices that this document has been updated three times, and that it has 14 incoming links. Although these features don’ t tell us anything about whether this document would match the subject of a query, they do give us some hints about the quality of the document.

A more concrete model of ranking

Notice that there are also feature functions that act on the query. The feature function $g_{\text{tropical}}(Q)$ evaluates to a large value because "tropical" is in the query. However, $g_{\text{barbs}}(Q)$ also has a small non-zero value because it is related to other terms in the query.

Inverted Indexes

Contents

- ✓Documents
- ✓Counts
- ✓Positions
- ✓Fields and Extents
- ✓Scores
- ✓Ordering

Inverted Indexes

An inverted index is the computational equivalent of the index found in the back of this textbook. Each index term is followed by a list of pages about that word.

If you want to know more about stemming, for example, you would look through the index until you found words starting with "s". Then, you would scan the entries until you came to the word "stemming." The list of page numbers there would lead you to Chapter 4.

Inverted Indexes

Similarly, an **inverted index** is organized by index term. The index is inverted because usually we think of words being a part of documents, but if we invert this idea, documents are associated with words.

Inverted Indexes

Index terms are often alphabetized like a traditional book index, but they need not be, since they are often found directly using a hash table.

Each index term has its own inverted list that holds the relevant data for that term.

Inverted Indexes

In this book, the relevant data is a list of page numbers. In a search engine, the data might be a list of documents or a list of word occurrences. Each list entry is called a **posting**, and the part of the posting that refers to a specific document or location is often called a **pointer**. Each document in the collection is given a unique number to make it efficient for storing document pointers.

Inverted Indexes

Indexes in books store more than just location information. For important words, often one of the page numbers is marked in boldface, indicating that this page contains a definition or extended discussion about the term.

Inverted files can also have extended information, where postings can contain a range of information other than just locations. By storing the right information along with each posting, the feature functions we saw in the last section can be computed efficiently.

Inverted Indexes

The page numbers in a book index are printed in ascending order.

Traditionally, inverted lists are stored the same way. These document-ordered lists are ordered by document number, which makes certain kinds of query processing more efficient and also improves list compression.

Some inverted files we will consider have other kinds of orderings.

Documents

The simplest form of an inverted list stores just the documents that contain each word, and no additional information.

and	1		only	2
aquarium	3		pigmented	4
are	3	4	popular	3
around	1		refer	2
as	2		referred	2

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

Four sentences from the Wikipedia entry

and	1
aquarium	3
are	3 4
around	1
as	2
both	1
bright	3
coloration	3 4
derives	4
due	3
environments	1
fish	1 2 3 4
fishkeepers	2
only	2
pigmented	4
popular	3
refer	2
referred	2
requiring	2
salt	1 4
saltwater	2
species	1
term	2
the	1 2
their	3
this	4

An inverted index for the documents

Notice that this index does not record the number of times each word appears; it only records the documents in which each word appears.

found	1
fresh	2
freshwater	1 4
from	4
generally	4
in	1 4
include	1
including	1
iridescence	4
marine	2
often	2 3
those	2
to	2 3
tropical	1 2 3
typically	4
use	2
water	1 2 4
while	4
with	2
world	1

Counts

and	1:1			
aquarium	3:1			
are	3:1	4:1		
around	1:1			
as	2:1			
both	1:1			
bright	3:1			
coloration	3:1	4:1		
derives	4:1			
due	3:1			
environments	1:1			
fish	1:2	2:3	3:2	4:2
fishkeepers	2:1			
only	2:1			
pigmented	4:1			
popular	3:1			
refer	2:1			
referred	2:1			
requiring	2:1			
salt	1:1	4:1		
saltwater	2:1			
species	1:1			
term	2:1			
the	1:1	2:1		
their	3:1			
this	4:1			

An inverted index, with word counts

This index looks similar to the previous one. However, each posting now has a second number. This second number is the number of times the word appears in the document.

found	1:1			
fresh	2:1			
freshwater	1:1	4:1		
from	4:1			
generally	4:1			
in	1:1	4:1		
include	1:1			
including	1:1			
iridescence	4:1			
marine	2:1			
often	2:1	3:1		
those	2:1			
to	2:2	3:1		
tropical	1:2	2:2	3:1	
typically	4:1			
use	2:1			
water	1:1	2:1	4:1	
while	4:1			
with	2:1			
world	1:1			

Counts

Compare with these two indexes:

For instance, consider the query “tropical fish”. Three sentences match this query: S_1, S_2, and S_3. The data in the document-based index gives us no reason to prefer any of these sentences over any other.

fish	1	2	3	4

tropical	1	2	3

Counts

Compare with these two indexes:

This small amount of additional data allows us to prefer S_2 over S_1 and S_3 for the query "tropical fish", since S_2 contains "tropical" twice and "fish" three times.

fish	1:2	2:3	3:2	4:2

tropical	1:2	2:2	3:1

Positions

and	1,15				
aquarium	3,5				
are	3,3	4,14			
around	1,9				
as	2,21				
both	1,13				
bright	3,11				
coloration	3,12	4,5			
derives	4,7				
due	3,7				
environments	1,8				
fish	1,2	1,4	2,7	2,18	2,23
			3,2	3,6	4,3
			4,13		
fishkeepers	2,1				

marine	2,22	
often	2,2	3,10
only	2,10	
pigmented	4,16	
popular	3,4	
refer	2,9	
referred	2,19	
requiring	2,12	
salt	1,16	4,11
saltwater	2,16	
species	1,18	
term	2,5	
the	1,10	2,4
their	3,9	
this	4,4	

An inverted index, with word positions

found	1,5	
fresh	2,13	
freshwater	1,14	4,2
from	4,8	
generally	4,15	
in	1,6	4,1
include	1,3	
including	1,12	
iridescence	4,9	

those	2,11				
to	2,8	2,20	3,8		
tropical	1,1	1,7	2,6	2,17	3,1
typically	4,6				
use	2,3				
water	1,17	2,14	4,12		
while	4,10				
with	2,15				
world	1,11				

Positions

When looking for matches for a query like “tropical fish”, the location of the words in the document is an important predictor of relevance.

Positions

We see that the word “tropical” is the first word in S_1, and “fish” is the second word in S_1, which means that S_1 must start with the phrase “tropical fish”. The word “tropical” appears again as the seventh word in S_1, but “fish” does not appear as the eighth word, so this is not a phrase match. In all, there are four occurrences of the phrase “tropical fish” in the four sentences.

tropical	1,1	1,7	2,6	2,17	3,1
fish	1,2	1,4	2,7	2,18	2,23
			3,2	3,6	4,3
			4,13		

Fields and Extents

Real documents are not just lists of words. They have sentences and paragraphs that separate concepts into logical units. Some documents have titles and headings that provide short summaries of the rest of the content. Special types of documents have their own sections; for example, every email contains sender information and a subject line. All of these are instances of what we will call **document fields**, which are sections of documents that carry some kind of semantic meaning.

Fields and Extents

```
<author>W. Bruce Croft</author>,
<author>Donald Metzler</author>, and
<author>Trevor Strohman</author>
```

Suppose you would like to find books by an author named **Croft Donald**. If you type the phrase query " croft donald" into a search engine, should this book match? The words "croft" and "donald" appear in it, and in fact, they appear next to each other. However, they are in two distinct author fields. This probably is not a good match for the query " croft donald", but the previous two methods for dealing with cannot make this kind of distinction.

Fields and Extents

An **extent** is a contiguous region of a document. We can represent these extents using word positions.

For example, if the title of a book started on the fifth word and ended just before the ninth word, we could encode that as (5,9).

Fields and Extents

```
<author>W. Bruce Croft</author>,
<author>Donald Metzler</author>, and
<author>Trevor Strohman</author>
```

For the author text shown earlier, we could write author: (1,4), (4,6), (7,9). The (1,4) means that the first three words (“W. Bruce Croft”) constitute the first author, followed by the second author(“Donald Metzler”), which is two words. The word “and” is not in an author field, but the next two words are, so the last posting is (7,9).

Fields and Extents

fish	1,2	1,4		2,7	2,18	2,23	3,2	3,6	4,3	4,13
title	1:(1,3)		2:(1,5)							4:(9,15)

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

At the very beginning of both lists, we see that document 1 has a title that contains the first two words (1 and 2, ending just before the third word). We know that this title includes the word "fish", because the inverted list for "fish" tells us that "fish" is the second word in document 1.

Fields and Extents

fish	1,2	1,4		2,7	2,18	2,23	3,2	3,6	4,3	4,13
title	1:(1,3)		2:(1,5)							4:(9,15)

If the user wants to find documents with the word “fish” in the title, document 1 is a match. Document 2 does not match, because its title ends just before the fifth word, but “fish” doesn’t appear until the seventh word. Document 3 apparently has no title at all, so no matches are possible. Document 4 has a title that starts at the ninth word (perhaps the document begins with a date or an author declaration), and it does contain the word “fish”. In all, this example shows two matching documents: 1 and 4.

Scores

We could make a list for “fish” that has postings like [(1:3.6), (3:2.2)], meaning that the total feature value for “fish” in document 1 is 3.6, and in document 3 it is 2.2. Presumably the number 3.6 came from taking into account how many times “fish” appeared in the title, in the headings, in large fonts, in bold, and in links to the document.

Scores

Storing scores like this both increases and decreases the system’ s flexibility. It increases flexibility because computationally expensive scoring becomes possible, since much of the hard work of scoring documents is moved into the index. However, flexibility is lost, since we can no longer change the scoring mechanism once the index is built.

Ordering

So far, we have assumed that the postings of each inverted list would be ordered by document number. Although this is the most popular option, this is not the only way to order an inverted list. An inverted list can also be ordered by score, so that the highest-scoring documents come first.

Compression

Contents

- ✓Entropy and Ambiguity
- ✓Delta Encoding
- ✓Bit-Aligned Codes
- ✓Byte-Aligned Codes
- ✓Compression in Practice
- ✓Looking Ahead
- ✓Skipping and Skip Pointers

Compression

There are many different ways to store digital information. Usually we make a simple distinction between **persistent** and **transient** storage.

Modern computers contain a **memory hierarchy**. At the top of the hierarchy we have memory that is tiny, but fast. The base consists of memory that is huge, but slow.

The performance of a search engine strongly depends on how it makes use of the properties of each type of memory.

Compression

Compression techniques are the most powerful tool for managing the memory hierarchy. The inverted lists for a **large** collection are themselves very large.

The index can be comparable in size to the document collection. **Compression** allows the same inverted list data to be stored in less space.

Compression

Benefit:

reduce disk or memory requirements, which would save money.

allow data to move up the memory hierarchy.

squeeze data closer together, which reduces seek times.

allow the processors to share memory bandwidth more efficiently

Compression

The space savings of compression comes at a cost: the processor must decompress the data in order to use it. Therefore, it isn’ t enough to pick the compression technique that can store the most data in the smallest amount of space.

In order to increase overall performance, we need to choose a compression technique that **reduces space** and is **easy to decompress**.

Compression

In this section, we consider only **lossless compression** techniques. Lossless techniques store data in less space, but without losing information.

There are also **lossy** data **compression** techniques, which are often used for video, images, and audio.

Entropy and Ambiguity

Compression techniques are based on probabilities. The fundamental idea behind compression is to represent common data elements with short codes while representing uncommon data elements with longer codes.

Entropy and Ambiguity

The inverted lists that are essentially lists of numbers, and without compression, each number takes up the same amount of space. Since some of those numbers are more frequent than others, if we encode the frequent numbers with short codes and the infrequent numbers with longer codes, we can end up with space savings.

Entropy and Ambiguity

For example, consider the numbers 0, 1, 2, and 3. We can encode these numbers using two binary bits. A sequence of numbers, like:

$$0, 1, 0, 3, 0, 2, 0$$

can be encoded in a sequence of binary digits:

$$00\ 01\ 00\ 10\ 00\ 11\ 00$$

Remember that the spaces in the code are only there for our convenience and are not actually stored.

Entropy and Ambiguity

Our first attempt at an encoding might be:

$$0\,01\,0\,10\,0\,11\,0$$

This encoding is, however, ambiguous, meaning that it is not clear how to decode it. If we add some different spaces, we arrive at a perfectly valid interpretation of this encoding:

$$0\,01\,01\,0\,0\,11\,0$$

Entropy and Ambiguity

When decoded, becomes:

$$0, 1, 1, 0, 0, 3, 0$$

Unfortunately, this isn’ t the data we encoded. The trouble is that when we see 010 in the encoded data, we can’ t be sure whether (0, 2) or (1, 0) was encoded.

Entropy and Ambiguity

The uncompressed encoding was not ambiguous. We knew exactly where to put the spaces because we knew that each number took exactly 2 bits. In our compressed code, encoded numbers consume either 1 or 2 bits, so it is not clear where to put the spaces.

Entropy and Ambiguity

To solve this problem, we need to restrict ourselves to unambiguous codes, which are confusingly called both **prefix codes** and **prefix-free codes**. An unambiguous code is one where there is only one valid way to place spaces in encoded data.

Entropy and Ambiguity

Let’ s fix our code so that it is unambiguous:

Number	Code
0	0
1	101
2	110
3	111

This results in the following encoding:

0 101 0 111 0 110 0

Delta Encoding

All of the coding techniques we will consider in this chapter assume that small numbers are more likely to occur than large ones. Remember that inverted list postings are typically ordered by document number. An inverted list without counts, for example, is just a list of document numbers, like these:

$$1, 5, 9, 18, 23, 24, 30, 44, 45, 48$$

Delta Encoding

$$1, 5, 9, 18, 23, 24, 30, 44, 45, 48$$

This fact allows us to encode the list of numbers by the differences between adjacent document numbers:

$$1, 4, 4, 9, 5, 1, 6, 14, 1, 3$$

This process is called delta encoding, and the differences are often called d-gaps.

Delta Encoding

The inverted lists for the words “entropy” and “who.” The word “who” is very common, so we expect that most documents will contain it. When we use delta encoding on the inverted list for “who,” we would expect to see many small d-gaps, such as:

$$1, 1, 2, 1, 5, 1, 4, 1, 1, 3, \ldots$$

Delta Encoding

By contrast, the word "entropy" rarely appears in text, so only a few documents will contain it. Therefore, we would expect to see larger d-gaps, such as:

$$109, 3766, 453, 1867, 992, \ldots$$

In general, we will find that inverted lists for frequent terms compress very well, whereas infrequent terms compress less well.

Bit-Aligned Codes

In all of the techniques we'll discuss, we are looking at ways to store small numbers in inverted lists (such as word counts, word positions, and delta encoded document numbers) in as little space as possible.

Bit-Aligned Codes

One of the simplest codes is the unary code. You are probably familiar with binary, which encodes numbers with two symbols, typically 0 and 1. A unary number system is a base-1 encoding, which means it uses a single symbol to encode numbers. Here are some examples:

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

Bit-Aligned Codes

In general, to encode a number k in unary, we output k 1s, followed by a 0. We need the 0 at the end to make the code unambiguous.

This code is very efficient for small numbers such as 0 and 1, but quickly becomes very expensive. For instance, the number 1023 can be represented in 10 binary bits, but requires 1024 bits to represent in unary code.

Bit-Aligned Codes

Now we know about two kinds of numeric encodings. Unary is convenient because it is compact for small numbers and is inherently unambiguous. Binary is a better choice for large numbers, but it is not inherently unambiguous.

Bit-Aligned Codes-Elias-γ codes

The Elias-γ (Elias gamma) code combines the strengths of unary and binary codes. To encode a number k using this code, we compute two quantities:

- $k_d = \lfloor \log_2 k \rfloor$
- $k_r = k - 2^{\lfloor \log_2 k \rfloor}$

Bit-Aligned Codes-Elias-γ codes

Suppose you wrote k in binary form. The first value, k_d, is the number of binary digits you would need to write. Assuming k > 0, the leftmost binary digit of k is 1. If you erase that digit, the remaining binary digits are k_r.

- $k_d = \lfloor \log_2 k \rfloor$
- $k_r = k - 2^{\lfloor \log_2 k \rfloor}$

- $k_d = \lfloor \log_2 k \rfloor$
- $k_r = k - 2^{\lfloor \log_2 k \rfloor}$

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	111111110 111111111

Bit-Aligned Codes-Elias-γ codes

The savings for large numbers is substantial. We can, for example, now encode 1023 in 19 bits, instead of 1024 using just unary code.

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. Therefore, $2\lfloor \log_2 k \rfloor + 1$ bits are required in all.

Bit-Aligned Codes-Elias-δ codes

- $k_{dd} = \lfloor \log_2(k_d + 1) \rfloor$
- $k_{dr} = (k_d + 1) - 2^{\lfloor \log_2(k_d+1) \rfloor}$

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

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

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

Bit-Aligned Codes-Elias-γ codes

Specifically, the Elias-γ code requires $\lfloor \log_2(\lfloor \log_2 k \rfloor + 1) \rfloor + 1$ bits for k_{dd} in unary, followed by $\lfloor \log_2(\lfloor \log_2 k \rfloor + 1) \rfloor$ bits for k_{dr} in binary, and $\lfloor \log_2 k \rfloor$ bits for k_r in binary. The total cost is approximately $2 \log_2 \log_2 k + \log_2 k$.

Byte-Aligned Codes

Like the other codes we have studied so far, the v-byte method uses short codes for small numbers and longer codes for longer numbers. However, each code is a series of bytes, not bits. So, the shortest v-byte code for a single integer is one byte.

Byte-Aligned Codes

The v-byte code is really quite simple. The low seven bits of each byte contain numeric data in binary. The high bit is a terminator bit. The last byte of each code has its high bit set to 1; otherwise, it is set to 0. Any number that can be represented in seven binary digits requires one byte to encode.

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

Space requirements for numbers encoded in v-byte

Sample encodings for v-byte

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

Compression in Practice

The compression techniques we have covered are used to encode inverted lists in real retrieval systems.

Consider just the inverted list for tropical:

$$(1,1)(1,7)(2,6)(2,17)(3,1)$$

In each pair, the first number represents the document and the second number represents the word position.

Compression in Practice

Add (2, 197) to the list: $(1,1)(1,7)(2,6)(2,17)(2,197)(3,1)$

We can group the positions for each document together so that each document has its own entry, (document, count, [positions]). Our example data now looks like this:

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

Interpret these numbers unambiguously, even if they were printed as follows: $1,2,1,7,2,3,6,17,197,3,1,1$

Compression in Practice

These are small numbers, but with delta encoding we can make them smaller.

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

We can delta-encode the positions as well:

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

Now we can remove the brackets and consider this inverted list as just a list of numbers:

$$1, 2, 1, 6, 1, 3, 6, 11, 180, 1, 1, 1$$

Compression in Practice

Since most of these numbers are small, we can compress them with v-byte to save space:

```
81 82 81 86 81 83 86 8B 01 B4 81 81 81
```

The 01 B4 is 180, which is encoded in two bytes. The rest of the numbers were encoded as single bytes, giving a total of 13 bytes for the entire list.

Looking Ahead

The best choice for a compression algorithm is tightly coupled with the state of modern CPUs and memory systems. For a long time, CPU speed was increasing much faster than memory throughput, so compression schemes with higher compression ratios became more attractive. However, the dominant hardware trend now is toward many CPU cores with lower clock speeds. Depending on the memory throughput of these systems, lower compression ratios may be attractive.

Skipping and Skip Pointers

For many queries, we don’ t need all of the information stored in a particular inverted list. Instead, it would be more efficient to read just the small portion of the data that is relevant to the query. Skip pointers help us achieve that goal.

Skipping and Skip Pointers

As a simple example, consider the following list of document numbers, uncompressed:

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

If we delta-encode this list, we end up with a list of d-gaps like this:

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

Skipping and Skip Pointers

We can then add some skip pointers for this list, using 0-based positions (that is, the number 5 is at position 0 in the list):

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

Suppose we try decoding using the skip pointer (34, 6). We move to position 6 in the d-gaps list, which is the number 2. We add 34 to 2, to decode document number 36.

Skipping and Skip Pointers

More generally, if we want to find document number 80 in the list, we scan the list of skip pointers until we find (52, 12) and (89, 15). 80 is larger than 52 but less than 89, so we start decoding at position 12. We find:

- $52 + 5 = 57$
- $57 + 23 = 80$

Skipping and Skip Pointers

If instead we were searching for 85, we would again start at skip pointer (52, 12):

- $52 + 5 = 57$
- $57 + 23 = 80$
- $80 + 9 = 89$

At this point, since $85 < 89$, we would know that 85 is not in the list.

Auxiliary Structures

The inverted file is the primary data structure in a search engine, but usually other structures are necessary for a fully functional system.

1. Vocabulary and statistics.
2. Documents, snippets, and external systems.

Vocabulary and statistics

An additional directory structure, called the vocabulary or lexicon, contains a lookup table from index terms to the byte offset of the inverted list in the inverted file.

Vocabulary and statistics

In many cases, this vocabulary lookup table will be small enough to fit into memory. In this case, the vocabulary data can be stored in any reasonable way on disk and loaded into a hash table at search engine startup. If the search engine needs to handle larger vocabularies, some kind of tree-based data structure, such as a B-tree, should be used to minimize disk accesses during the search process.

Documents, snippets, and external systems

The search engine, as described so far, returns a list of document numbers and scores. However, a real user-focused search engine needs to display textual information about each document, such as a document title, URL, or text summary. In order to get this kind of information, the text of the document needs to be retrieved.

Index Construction

Contents

- ✓Simple Construction
- ✓Merging
- ✓Parallelism and Distribution
- ✓Update

Simple Construction

```
procedure BUILDINDEX(D)                          ▷ D is a set of text documents
    I ← HashTable()                              ▷ Inverted list storage
    n ← 0                                        ▷ Document numbering
    for all documents d ∈ D do
        n ← n + 1
        T ← Parse(d)                             ▷ Parse document into tokens
        Remove duplicates from T
        for all tokens t ∈ T do
            if I_t ∉ I then
                I_t ← Array()
            end if
            I_t.append(n)
        end for
    end for
    return I
end procedure
```

Pseudocode for a simple indexer

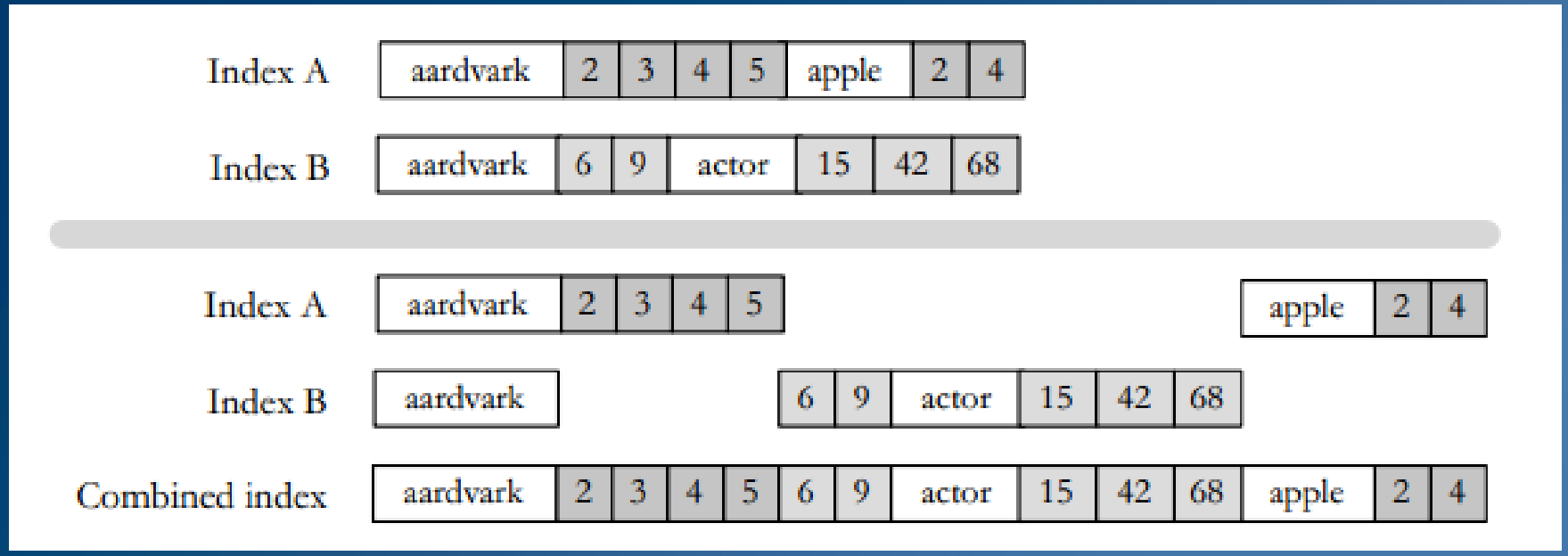
Simple Construction

The process involves only a few steps. A list of documents is passed to the BuildIndex function, and the function parses each document into tokens, These tokens are words, perhaps with some additional processing, such as downcasing or stemming. Then, for each token, the function determines whether a new inverted list needs to be created in I, and creates one if necessary. Finally, the current document number, n, is added to the inverted list.

Simple Construction

It is limited in two ways. First, it requires that all of the inverted lists be stored in memory, which may not be practical for larger collections. Second, this algorithm is sequential, with no obvious way to parallelize it.

Merging



An example of index merging. The first and second indexes are merged together to produce the combined index.

Merging

By definition, it is not possible to hold even two of the partial index files in memory at one time, so the input files need to be carefully designed so that they can be merged in small pieces.

This figure shows an example of this kind of merging procedure. Even though this figure shows only two indexes, it is possible to merge many at once.

Merging

This merging strategy also shows a possible parallel indexing strategy. If many machines build their own partial indexes, a single machine can combine all of those indexes together into a single, final index.

Parallelism and Distribution

The traditional model for search engines has been to use a single, fast machine to create the index and process queries. Tis is still the appropriate choice for a large number of applications, but it is no longer a good choice for the largest systems. Instead, for these large systems, it is increasingly popular to use many inexpensive servers together and use distributed processing software to coordinate their activities. MapReduce is a distributed processing tool that makes this possible.

Parallelism and Distribution

Two factors have forced this shift. First, the amount of data to index in the largest systems is exploding.

The second factor is simple economics. The incredible popularity of personal computers has made them very powerful and inexpensive. In contrast, large computers serve a very small market, and therefore have fewer opportunities to develop economies of scale.

Parallelism and Distribution

Inexpensive servers have a few disadvantages when compared to mainframes. First, they are more likely to break, and the likelihood of at least one server failure goes up as you add more servers. Second, they are difficult to program.

Update

So far, we have assumed that indexing is a batch process. Tis means that a set of documents is given to the indexer as input, the indexer builds the index, and then the system allows users to run queries. In practice, most interesting document collections are constantly changing. At the very least, collections tend to get bigger over time; every day there is more news and more email.

Update

We can solve the problem of update with two techniques: index merging and result merging.

If the index is stored in memory, there are many options for quick index update. Typically the index is stored on a disk. Inserting data in the middle of a file is not supported by any common file system, so direct disk-based update is not straightforward.

Update

Index merging is a reasonable update strategy when index updates come in large batches, perhaps many thousands of documents at a time. For single document updates, it isn’ t a very good strategy. For these small updates, it is better to just build a small index for the new data, but not merge it into the larger index. Queries are evaluated separately against the small index and the big index, and the result lists are merged to find the top k results.

Update

Result merging solves the problem of how to handle new documents: just put them in a new index.

The common solution is to use a deleted document list. During query processing, the system checks the deleted document list to make sure that no deleted documents enter the list of results shown to the user.

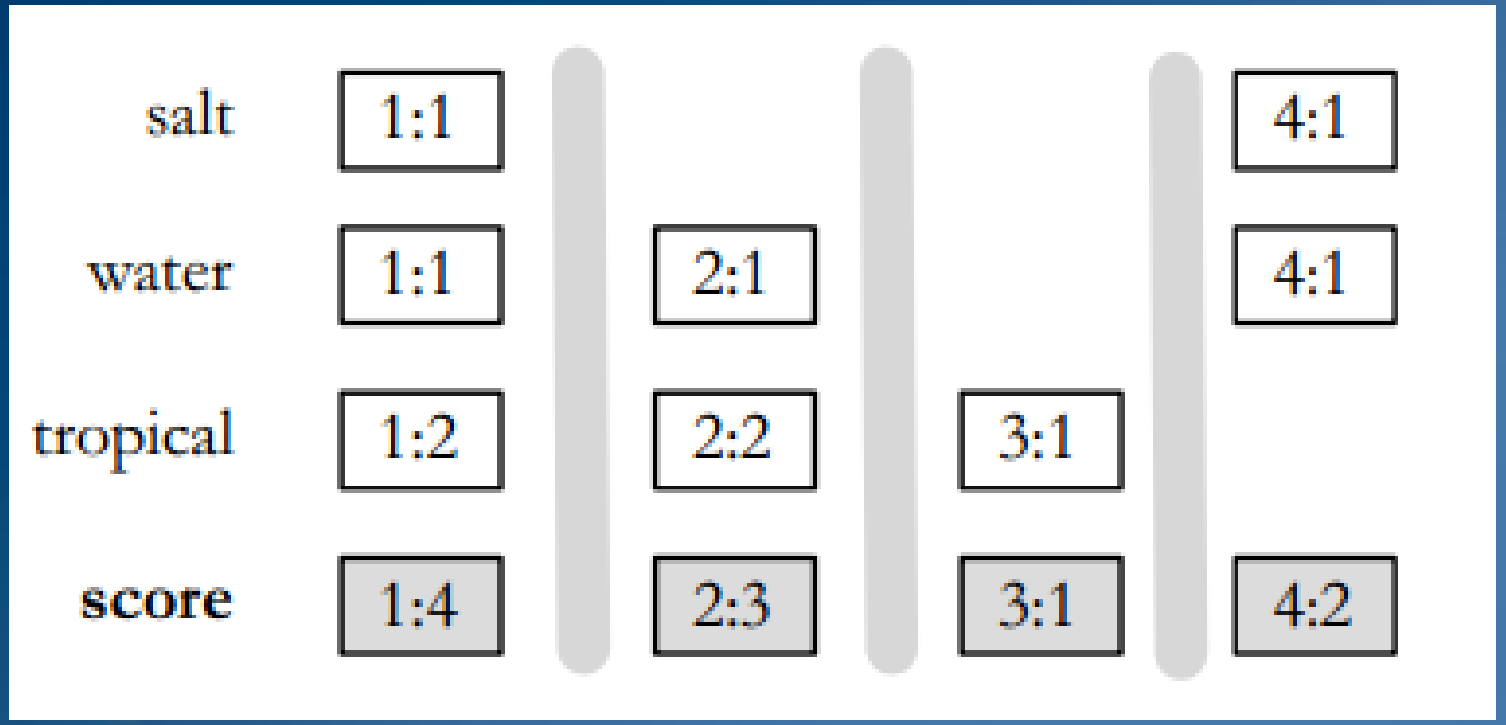
Query Processing

Contents

- ✓Document-at-a-time Evaluation
- ✓Term-at-a-time Evaluation
- ✓Optimization Techniques
- ✓Structured Queries
- ✓Distributed Evaluation
- ✓Caching

Query Processing

Once an index is built, we need to process the data in it to produce query results. Even with simple algorithms, processing queries using an index is much faster than it is without one. However, clever algorithms can boost query processing speed by Ten to a hundred times over the simplest versions.


salt
water
tropical
score
1:1
1:1
1:2
1:4
2:1
2:2
2:3
3:1
3:1
4:1
4:1
4:2

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all documents d ∈ I do
        s_d ← 0
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q) f_i(l_i)          ▷ Update the document score
            end if
            l_i.movePastDocument( d )
        end for
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

Document-at-a-time Evaluation

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all documents d ∈ I do
        s_d ← 0
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q) f_i(l_i)          ▷ Update the document score
            end if
            l_i.movePastDocument( d )
        end for
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

The parameters are Q, the query; I, the index; f and g, the sets of feature functions; and k, the number of documents to retrieve.

Document-at-a-time Evaluation

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all documents d ∈ I do
        s_d ← 0
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q) f_i(l_i)          ▷ Update the document score
            end if
            l_i.movePastDocument( d )
        end for
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

For each word w_i in the query, an inverted list is fetched from the index. These inverted lists are assumed to be sorted in order by document number. The Inverted List object starts by pointing at the first posting in each list. All of the fetched inverted lists are stored in an array, L.

Document-at-a-time Evaluation

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all documents d ∈ I do
        s_d ← 0
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q) f_i(l_i)          ▷ Update the document score
            end if
            l_i.movePastDocument( d )
        end for
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

In the main loop, the function loops once for each document in the collection. At each document, all of the inverted lists are checked.

At the end of each document loop, a new document score has been computed and added to the priority queue R.

Document-at-a-time Evaluation

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all documents d ∈ I do
        s_d ← 0
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q) f_i(l_i)          ▷ Update the document score
            end if
            l_i.movePastDocument( d )
        end for
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

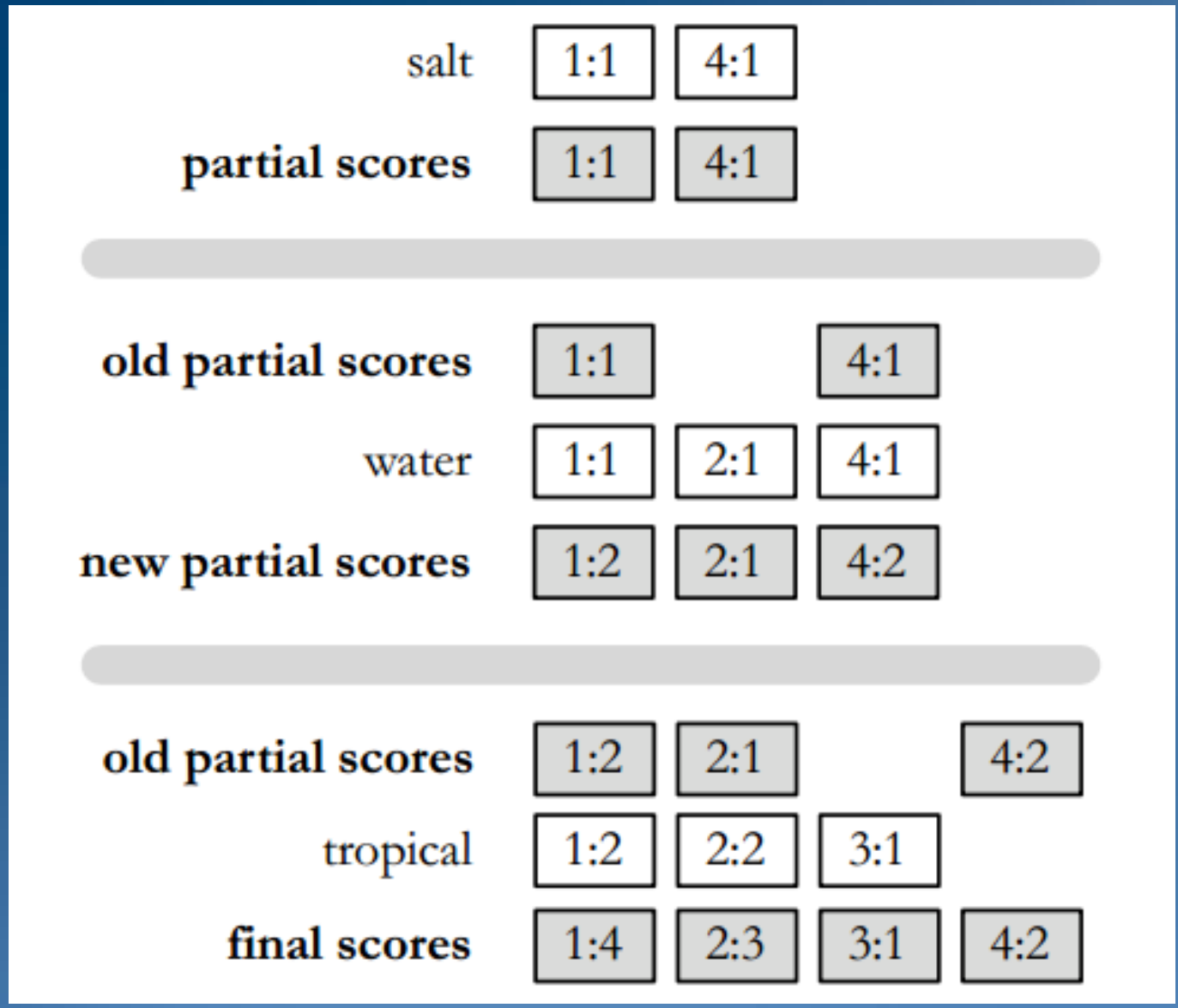
the priority queue R only needs to hold the top k results at any one time. If the priority queue ever contains more than k results, the lowest-scoring documents can be removed until only k remain, in order to save memory.

Document-at-a-time Evaluation

```
procedure DocumentAtATimeRetrieval(Q, I, f, g, k)
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all documents d ∈ I do
        s_d ← 0
        for all inverted lists l_i in L do
            if l_i.getCurrentDocument() = d then
                s_d ← s_d + g_i(Q) f_i(l_i)        ▷ Update the document score
            end if
            l_i.movePastDocument( d )
        end for
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

The primary benefit of this method is its frugal use of memory. The only major use of memory comes from the priority queue, which only needs to store k entries at a time.

114 Term-at-a-time Evaluation



Term-at-a-time Evaluation

```
procedure TermAtATimeRetrieval(Q, I, f, g k)
    A ← HashTable()
    L ← Array()
    R ← PriorityQueue(k)
    for all terms w_i in Q do
        l_i ← InvertedList(w_i, I)
        L.add( l_i )
    end for
    for all lists l_i ∈ L do
        while l_i is not finished do
            d ← l_i.getCurrentDocument()
            A_d ← A_d + g_i(Q) f(l_i)
            l_i.moveToNextDocument()
        end while
    end for
    for all accumulators A_d in A do
        s_d ← A_d                          ▷ Accumulator contains the document score
        R.add( s_d, d )
    end for
    return the top k results from R
end procedure
```

Document-at-a-time Evaluation

In practice, neither the document-at-a-time nor term-at-a-time algorithms are used without additional optimizations. These optimizations dramatically improve the running speed of the algorithms, and can have a large effect on the memory footprint.

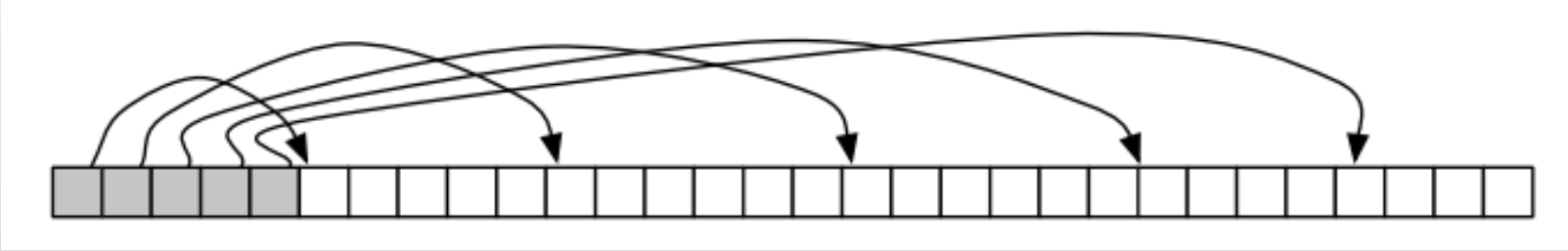
Optimization Techniques

There are two main classes of optimizations for query processing. The first is to read less data from the index, and the second is to process fewer documents.

When using feature functions that are particularly complex, focusing on scoring fewer documents should be the main concern. For simple feature functions, the best speed comes from ignoring as much of the inverted list data as possible.

List skipping

This kind of forward skipping is by far the most popular way to ignore portions of inverted lists. More complex approaches (for example, tree structures) are also possible but not frequently used.



Skip pointers in an inverted list. The gray boxes show skip pointers, which point into the white boxes, which are inverted list postings.

List skipping

Although it might seem that list skipping could save on disk accesses, in practice it rarely does. Modern disks are much better at reading sequential data than they are at skipping to random locations.

Even so, skipping is still useful because it reduces the amount of time spent decoding compressed data that has been read from disk, and it dramatically reduces processing time for lists that are cached in memory.

Conjunctive processing

The simplest kind of query optimization is conjunctive processing. By conjunctive processing, we just mean that every document returned to the user needs to contain all of the query terms.

With short queries, conjunctive processing can actually improve effectiveness and efficiency simultaneously. In contrast, search engines that use longer queries, such as entire paragraphs, will not be good candidates for conjunctive processing.

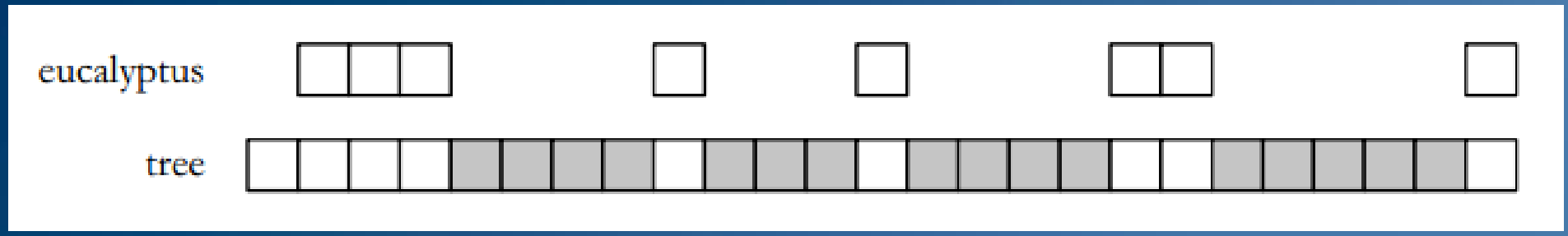
Conjunctive processing

Conjunctive processing works best when one of the query terms is rare, as in the query “fish locomotion”. The word “fish” occurs about 100 times as often as the word “locomotion”. Since we are only interested in documents that contain both words, the system can skip over most of the inverted list for “fish” in order to find only the postings in documents that also contain the word “locomotion”.

Threshold methods

Remember that k is the number of results requested by the user, and for many search applications this number is something small, such as 10 or 20. Because of this small value of k, most documents in the inverted lists will never be shown to the user. Threshold methods focus on this k parameter in order to score fewer documents.

MaxScore



MaxScore retrieval with the query “eucalyptus tree”. The gray boxes indicate postings that can be safely ignored during scoring.

MaxScore

Suppose that the indexer computed the largest partial score in the “tree” list, and that value is called μtree. Tis is the maximum score (hence MaxScore) that any document that contains just this word could have.

Early termination

In term-at-a-time systems, we can terminate processing by simply ignoring some of the very frequent query terms.

In document-at-a-time systems, early termination means ignoring the documents at the very end of the inverted lists.

List ordering

If the document numbers are assigned randomly, this means that the document sort order is random. The net effect is that the best documents for a query can easily be at the very end of the lists.

Since these lists can be long, it makes sense to consider a more intelligent ordering.

List ordering

One way to improve document ordering is to order documents based on document quality, as we discussed in the last section. There are plenty of quality metrics that could be used, such as PageRank or the total number of user clicks.

List ordering

Another option is to order each list by partial score. For instance, for the "food" list, we could store documents that contain many instances of the word "food" first. For a "dog" list, we could store pages about dogs (i.e., containing many instances of "dog") first.

Structured Queries

Structured queries are queries written in a query language, which allows you to change the features used in a query and the way those features are combined. The query language is not used by normal users of the system.

Instead, a query translator converts the user’ s input into a structured query representation.

Distributed Evaluation

A single modern machine can handle a surprising load, and is probably enough for most tasks. However, dealing with a large corpus or a large number of users may require using more than one machine.

Distributed Evaluation

The general approach to using more than one machine is to send all queries to a director machine. The director then sends messages to many index servers, which do some portion of the query processing. The director then organizes the results of this process and returns them to the user.

Distributed Evaluation

Another distribution method is called term distribution. In term distribution, a single index is built for the whole cluster of machines. Each inverted list in that index is then assigned to one index server. Therefore, in most cases the data to process a query is not stored all on one machine.

Distributed Evaluation

The easiest distribution strategy is called document distribution. In this strategy, each index server acts as a search engine for a small fraction of the total document collection. The director sends a copy of the query to each of the index servers, each of which returns the top k results, including the document scores for these results.

Caching

Broadly speaking, caching means storing something you might want to use later.

Caching is perfectly suited for search engines. Queries and ranked lists are small, meaning it doesn’ t take much space in a cache to store them.

Recent research suggests that when memory space is tight, caching should focus on the most popular queries, leaving plenty of room to cache the index.

Caching

When using caching systems, it is important to guard against stale data. Caching works because we assume that query results will not change over time, but eventually they do. Cache entries need acceptable timeouts that allow for fresh results.