Announcements

Homework 2 is out today, due two weeks from now.

Group project demo slot sign up by December 5.

- See course homepage for sign up form, first come first serve

Topics for the project

- You are not restricted by the advanced topics from the class
- Previous years' common topics include
  - Large scale indexing using AWS, Google Cloud, etc.
  - Real time search over social feeds or news
  - Multimedia retrieval
  - Personalized search or query suggestion
- Regardless of the topic, the project should be built on top of the homework, exceptions must obtain our approval first.

Brief Review of Last Three Lectures

Evaluation

- Ranking Model
- Query Model
Evaluation

Batch study
• Repeatable experiments
• High quality judgments
• Synthetic environment

Production test
• Repeatable experiments
• Real environment
• Imprecise judgments

Precision and Recall

PR Curve

AUC

F-Measure
Evaluation

**Average Precision and MAP**

- [ ] 1.0
- [ ] 1.0
- [ ] 0.5
- [ ] 0.4

**DCG and nDCG**

- [ ] DCG = 1.00
- [ ] DCG = 0.88
- [ ] DCG = 0.59
- [ ] DCG = 0.49

Ranking Model

**Vector Space Model**

- Queries and documents are represented as vectors of weighted terms
- Using vector similarity measure the similarity between the query and the document
- Topicality driven model that does not require signals from user behaviors

**Page Rank**

- Documents are modeled as a graph via links
- Using the matrix transition probability to capture the quality signals independent of the content
- Simulates user behaviors before there is large amount of user logs

**Learning to Rank (heavily uses large scale user data)**

Query Model

**Beyond Explicit Query Specification**

- Synonyms: e.g., [NYU]
- Implicit constraints: e.g., [restaurants]
- Session: e.g., [Italian restaurants], [what about french]
- Personalization: e.g., [apple]

**Query Expansion**

- Corpus driven
- Log driven
Why Do We Need Index for Search?

Making Search Faster
- Critical in the whole search experience
  - Faster search response time → users will not be afraid of making mistakes b/c the cost of issuing another query is low
  - Eye blink: 400ms

Driven by the Ranking Models
- Users are issuing keyword based queries, thus the index needs to support fast lookup and scoring for documents based on keywords
- Different ranking models demand different indexing structures
  - Example 1: search by image or song
  - Example 2: structured queries
Abstract Model of Ranking

\[ R(q, D) = \sum_i g_i(q) f_i(D) + \sum_j h_j(D) \]

g_i(q) \cdot f_i(D) \text{ measure the topical matching}

h_j(D) \text{ measure the document quality}

Both sets of signals are combined in the ranking models

Terms as Topical Features

Intuitively, a document containing more query terms are more likely to be relevant to the query.

Thus, we want to organize documents according to the terms, why?

• When you search over billions of documents, most of the documents will have no match for the query topical features, i.e., their \( f(D) \) will all be zero.

Most common indexing data structure: inverted index

• Also called inverted file, inverted lists, or postings lists

• And you have probably used it if you've used a textbook!
Example Corpus

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.

A Simple Inverted Index

word → a list of documents
Each entry in the list is called a posting, hence postings list
All lists are typically stored in a single file called inverted file
• The inverted file can also be distributed
Assuming postings are sorted by docid, and the lists are implemented as arrays.

Accessing the Inverted Index

Goal is to quickly find documents containing the words in a query to feed into the ranking function

Access methods over the index:
• num_docs(): returns the number of documents in the corpus
• doc_info(docid): retrieve the document-level (i.e., quality) features of the document.
• term_info(term): retrieve the corpus-wide term features
• NextDocument(query, docid): retrieve the next “candidate” document

Access methods over a single postings list:
• next(term, docid): the next document after docid
• (alternatively) docs(term): the list of documents that word appears in, often not directly used because the result can be too large
• (optionally) prev, first, last
A Simple NextDocument Implementation

Finding the next document containing two words.
In HW2, you will need to handle more than two words.

NextDocument(w1, w2, did)
1. did2 = next(w2, did1);
2. did1 = next(w1, did1);
3. if did1 == ∞ or did2 == ∞ then
   4. return ∞;
5. end if
6. if did1 == did2 then
   7. return did1; // found!
8. end if
9. did1 = max(did1, did2);
10. return NextDocument(w1, w2, did1) - 1

Implementing next

Linear Search: \( P \) is the posting list, \( l \) is the largest position of \( P \), \( C \) is the cached index offset from previous call

\[
\text{next}(t, \text{current}) =
\begin{align*}
1 & \text{ if } l = 0 \text{ or } P[l] \leq \text{current} \text{ then} \\
2 & \quad \text{return } \infty \\
3 & \text{ if } P[l] > \text{current} \text{ then} \\
4 & \quad c_t \leftarrow 1 \\
5 & \quad \text{return } P[c_t] \\
6 & \text{ if } c_t > 1 \text{ and } P[c_t - 1] > \text{current} \text{ then} \\
7 & \quad c_t \leftarrow 1 \\
8 & \text{ while } P[c_t] \leq \text{current} \text{ do} \\
9 & \quad c_t \leftarrow c_t + 1 \\
10 & \text{return } P[c_t]
\end{align*}
\]

Implementing next

Binary Search: no need to cache offset from previous call

\[
\text{next}(t, \text{current}) =
\begin{align*}
1 & \text{ if } l = 0 \text{ or } P[l] \leq \text{current} \text{ then} \\
2 & \quad \text{return } \infty \\
3 & \text{ if } P[l] > \text{current} \text{ then} \\
4 & \quad \text{return } P[l] \\
5 & \text{ binarySearch}(t, \text{low, high, current}) = \\
6 & \text{ while } \text{high} - \text{low} > 1 \text{ do} \\
7 & \quad \text{mid} = \frac{\text{low + high}}{2} \\
8 & \quad \text{if } P[\text{mid}] \leq \text{current} \text{ then} \\
9 & \quad \text{low} \leftarrow \text{mid} \\
10 & \quad \text{else} \\
11 & \quad \text{high} \leftarrow \text{mid} \\
12 & \text{return } \text{high}
\end{align*}
\]
Which Access Method is Better: Linear or Binary?

“the courant”
- the: 25 billion results
- courant: 90 million results

“super bowl”
- super: 400 million results
- bowl: 400 million results

Linear Search
- Suitable when both words are equally frequent in the corpus

Binary Search
- Excellent when one of the words are much less frequent

Implementing next

Galloping Search (combines linear and binary)

```
next (l, current) :=
1 if l = 0 or P[l][j] ≤ current then
2    return ∞
3    c1 := 1
4    r := P[l][j]
5    if c1 ≥ 1 and P[c1−1] ≤ current then
6        lo := c1−1
7    else
8        lo := 1
9        jump := 1
10       high := low + jump
11       while high < l and P[high] ≤ current do
12         low := high
13         jump := 2 * jump
14       high := low + jump
15       if high > l then
16           lo := high
17           high := low
18           P[lo] := next (lo, current)
19       return P[r]
```

Beyond Document Only Inverted Index

Document only inverted index is good enough to find documents containing words in the query

However:
- Frequency of words in a document is essential for ranking
- Proximity of the words in the document is important
- Users sometimes want to do exact phrase search

None can be handled by the document only inverted index
Access Methods with Occurrences Inverted Index

Now we need new access methods to get the word positions.

- **next_pos**(term, docid, pos): the next occurrence of the term in
docid at or after pos
- (optional) **prev_pos**, **first_pos**, **last_pos**

Verify the document containing the term first, and if verified,
find the next position.
Simple Phrase Search using `next_pos`

Finding the next document a phrase of two words.
In HW2, you will need to handle longer phrases.

```
NextPhrase(w_i, w_j, docid, pos)
  1. docid_next = NextDocument(v_i, w_j, docid - 1);
  2. If docid_next # docid then
     3. return no
  4. end if
  5. pos_1 = next_pos(v_i, docid, pos);
  6. pos_2 = next_pos(v_j, docid, pos);
  7. If pos_1 == oo or pos_2 == oo then
     8. return no
  9. end if
 10. If pos_1 + 1 == pos_2 then
    11. return pos_1
 12. end if
 13. return NextPhrase(v_i, v_j, docid, max(pos_1, pos_2) - 1)
```

Indexing Structures within a Document

Documents, especially Web pages, often have internal structures.

- title, navigational menu, main body, etc.
- A query with words all matching inside the title is more relevant than some words in title and others in the body.
- Such structures can be incorporated in many ways
  - Using extent lists (see below): more extendable, but more lists to process

<table>
<thead>
<tr>
<th>Position</th>
<th>Title</th>
<th>&lt;1.1&gt;</th>
<th>&lt;1.2&gt;</th>
<th>&lt;2.1&gt;</th>
<th>&lt;2.2&gt;</th>
<th>&lt;2.3&gt;</th>
<th>&lt;2.4&gt;</th>
<th>&lt;2.5&gt;</th>
<th>&lt;2.6&gt;</th>
<th>&lt;2.7&gt;</th>
<th>&lt;2.8&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- As attributes to the postings: faster, but can only have a few attributes

Outline of Today’s Lecture

- Indexing Data Structure and Access Methods
- Compression
- Construction
- Query Processing
Practical Implementation of Posting Lists

fish: \[(1,2), (1,4), (2,7), (2,18), (2,23), (3,2), (3,6), (4,3), (4,13)\]
can be converted into:

fish: (1,2, [2,4]), (2,3, [7,18,23]), (3,2, [2,6]), (4,2, [3,13])

and, you don’t need the brackets and parentheses!

fish: 1, 2, 2, 4, 2, 3, 7, 18, 23, 3, 2, 2, 6, 4, 2, 3, 13

Each postings list can be considered as a long list of numbers, residing on disk.
And the list is very loooooooooong!

Why Compression?

Inverted index is usually huge

• Comparable in size with the corpus itself, or even bigger

It’s costly to read data from disk into memory

Compression makes it possible to keep more postings lists in memory and speed things up significantly, at the cost of having to decompress them when they are used

Index compression is thus finding an optimal balance between the costs of reading the postings lists in compressed format and decompressing them for ranking

• The most efficient compression mechanism is not necessarily the most appropriate

Index Compression

What do we want to compress? Numbers!

An Integer is 4 or 8 bytes, we can shrink that using bit-aligned codes:

• Elias-γ Code

• Elias-δ Code
Elias-γ Code

\[ k = 2^d + r \]

Encode \( d = \text{floor}(\log_2 k) \) as unary, with ending 0

Encode \( r = k - 2^d \) as binary, filled to \( d \) bits

<table>
<thead>
<tr>
<th>Number ( (k) )</th>
<th>( k )</th>
<th>( d )</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>11010</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>7</td>
<td>110111</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>0</td>
<td>011100000</td>
</tr>
<tr>
<td>255</td>
<td>7</td>
<td>127</td>
<td>1111111111111</td>
</tr>
<tr>
<td>1023</td>
<td>9</td>
<td>511</td>
<td>1111111111111111</td>
</tr>
</tbody>
</table>

In-Class Exercise

Encode 18 in Elias-γ code

\[ 18 = 2^4 + 2 \rightarrow 111100010, \text{9 bits < 32} \]

Encode 1024×1024

\[ 1024×1024 = 2^{20} + 0 \rightarrow 111...110000...000, \text{41 bits > 32!} \]

Decode 1110101101

\[ 2^3 + 5 = 13, 2^1 + 1 = 3 \rightarrow 13, 3 \]

Elias-δ Code

\[ k = 2^{d-2^{d_r}} + r \]

Compute \( d \) and \( r \) as in Elias-γ code

Encode \( d_r = \text{floor}(\log_2(d+1)) \) as unary, with ending 0

Encode \( r = k - 2^d \) as binary, filled to \( d \) bits

<table>
<thead>
<tr>
<th>Number ( (k) )</th>
<th>( k )</th>
<th>( d )</th>
<th>( d_r )</th>
<th>( r )</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1001</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>10 10</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>11000 111</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1100100000</td>
</tr>
<tr>
<td>255</td>
<td>7</td>
<td>127</td>
<td>3</td>
<td>0</td>
<td>1100000111111</td>
</tr>
<tr>
<td>1023</td>
<td>9</td>
<td>511</td>
<td>3</td>
<td>2</td>
<td>1100010111111111</td>
</tr>
</tbody>
</table>
In-Class Exercise

Encode 18 in Elias-δ code
18 = 2^4 + 2, 4 + 1 = 2^2 + 1 \Rightarrow 110010010, 9 bits < 32

Encode 1024x1024
1024x1024 = 2^20 + 0, 20 + 1 = 2^4 + 5 \Rightarrow 111100100000\ldots000, 29 bits < 32

Decode the first number from 11010000100\ldots
2^2 + 2 = 5 + 1; 2^5 + 1 = 33

Byte-Aligned Code
More common in practice since machines read multiples of bytes at a time
Most common: \(v\)-byte, a.k.a. variable length integer

Idea:
• Use only the bytes necessary to encode a number
• Small number should use fewer bytes
• Requires 1 bit per byte to indicate whether the current byte is the end of the number: 1 means last byte, 0 means more bytes after
• What's the maximum integer a 4 byte \(v\)-byte integer can store?
• \(2^{28} - 1\), because 4 bits are used to store the indicator.

Byte-Aligned Code

<table>
<thead>
<tr>
<th>(k)</th>
<th>Number of bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k &lt; 2^7)</td>
<td>1</td>
</tr>
<tr>
<td>(2^7 \leq k &lt; 2^{14})</td>
<td>2</td>
</tr>
<tr>
<td>(2^{14} \leq k &lt; 2^{21})</td>
<td>3</td>
</tr>
<tr>
<td>(2^{21} \leq k &lt; 2^{28})</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(k)</th>
<th>Binary Code</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 0000001</td>
<td>81</td>
</tr>
<tr>
<td>6</td>
<td>1 0000110</td>
<td>86</td>
</tr>
<tr>
<td>127</td>
<td>1 1111111</td>
<td>FF</td>
</tr>
<tr>
<td>128</td>
<td>0 0000001</td>
<td>01 80</td>
</tr>
<tr>
<td>129</td>
<td>0 0000001</td>
<td>01 81</td>
</tr>
<tr>
<td>130</td>
<td>0 0000001</td>
<td>01 82</td>
</tr>
<tr>
<td>20000</td>
<td>0 0000001</td>
<td>0 011000 0100000</td>
</tr>
</tbody>
</table>
Revisiting the Postings List for **fish**

$$\text{fish: (1,2,[2,4]), (2,3,\{7,18,23\}), (3,2,\{2,6\}), (4,2,\{3,13\})}$$

The Web have billions of documents, their docids will be large.

Same for the word position offsets.

Large numbers are not easily compressed!

---

**Delta Encoding**

$$\text{fish: (1,2,[2,4]), (2,3,\{7,18,23\}), (3,2,\{2,6\}), (4,2,\{3,13\})}$$

Both the docids and the occurrences are monotonically increasing, use *delta* to record just the difference

$$\text{fish: (1,2,[2,2]), (1,3,\{7,11,5\}), (1,2,[2,4]), (1,2,\{3,10\})}$$

Can then be encoded using v-byte as (in Hex)

$$\text{fish: 81 \ 82 \ 82 \ 01 \ 84 \ 82 \ 81 \ 83 \ 87 \ 8B \ 85, 81 \ 82 \ 82 \ 84, 81 \ 82 \ 83 \ 8A}$$

17 bytes instead of 68 bytes.

---

**In-Class Exercise**

Decode 81 82 82 01 84 82 81 83

$$(1, 2, [2, 132]), (2, 1, [3]) \rightarrow (1, 2, [2, 134]), (3, 1, [3])$$
Revisiting the Access Methods
Compression with delta encoding is nice, but it also destroys the non-linear access methods.

Why?
The actual docid can only be calculated by visiting all of the previous delta docids.

Example: 1 1 2 1, the third document is in fact document 4

Skip Pointer
Additional data structure to allow us perform non-linear search over compressed delta-encoded postings lists

\[(d, p)\]
- \(p\): the byte offset in the postings list where a posting starts
- \(d\): the docid right before the posting at \(p\)

Skip Pointer Example
fish: (1,2,2,4), (2,3,7,18,23), (3,2,2,6), (4,2,3,13)
fish: (1,2,2,2), (1,3,7,11,5), (1,2,2,4), (1,2,3,10)
fish: 81 82 82 82, 81 83 87 88 85, 81 82 82 84, 81 82 83 8A
Three possible skip pointers: (1, 4), (2, 9), (3, 13)
In practice, you want to skip hundreds of bytes.
Using Skip Pointer

Skip pointers are usually prepended to the beginning of the postings list.

Implementing next(word, docid)

1. Linear scan or binary search over the list of skip pointers
2. Locate the skip pointer with the largest docid that is still smaller than the given docid
3. Jump to the byte offset provided by the skip pointer
4. Linear search until the desired document is found

Index Construction

In-Memory

Merging

Distributed

In-Memory Index Construction (w/ Occurrences)

```
BuildInMemoryD(docid)
1. i = HashTable<string, List<Pair<int, int>>>
2. n = -1 // Document ID
3. for all d ∈ D do
4.   T = ParseAndTokenize(s), // Lecture 4
5.   y = -1; // Occurrence Position
6.   for all t ∈ T do
7.     if y < n then
8.       i = get(i)
9. if i == null then
10.      l = List<Pair<int, int> unl
11.     i.put(y, l)
12.     end if
13.     L.append(Pair<int, int>(y, y))
14.     end for
15.     for all l ∈ l do
16.       R = ConvertAndCompress(l); // produce byte[]
17.       WriteToDisk(R)
18.     end for
19.   end for
20. end for
```
Index Construction Using Merging

The In-Memory approach quickly fails as the corpus grows.

We need an approach that is not memory bound

• Build partial indices from subsets of documents
• Write those partial indices out to disk as needed
  • Be smart about this: e.g., write all postings lists for tokens starting with “a” to a single file, and sort them alphabetically.
  • This will help the merging process.
• Merge the partial indices into the full index
  • The merge process itself must not be memory bound
  • When the corpus is super large, this may become a multi-round process

Index Construction via Merging

<table>
<thead>
<tr>
<th>Index Construction via Merging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index A</td>
</tr>
<tr>
<td>apple</td>
</tr>
<tr>
<td>Index B</td>
</tr>
<tr>
<td>school</td>
</tr>
<tr>
<td>index</td>
</tr>
</tbody>
</table>

| Index A                        |
| apple                           |
| Index B                        |
| school                         |

| Combined index                  |
| apple                           |
| school                         |

| Index A                        |
| apple                           |
| Index B                        |
| school                         |

Construct Web Index

You simply can’t index the Web on a single machine!

Must use distributed methods, to be discussed later.
Outline of Today’s Lecture

Indexing Data Structure and Access Methods

Compression

Construction

Query Processing

Processing is not Ranking, but Closely Connected

The ProcessQuery function below is the simplest query processing + ranking mechanism.

• How to implement NextDocument(q, docid) is part of your HW2.
• The ranking model is binary: the first set of $k$ documents with both query terms are returned.

```
ProcessQuery(q, k)
1: D = List()
2: docid = NextDocument(q, -1)
3: while docid ≠ ∞ and |D| < k do
4:   D.append(docid)
5:   docid = NextDocument(q, docid)
6: end while
7: return D
```

Ranking Model in Real World is More Complex

Query Processing is about gathering the necessary information for the ranker to do its job well.

Two Main Approaches

• Document-at-a-time
• Term-at-a-time
Document-at-a-time Query Processing
Processing all postings lists for the query terms together gradually and score the documents one by one.

<table>
<thead>
<tr>
<th></th>
<th>salt</th>
<th>water</th>
<th>tropical</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>freq</td>
<td>1:1</td>
<td>2:1</td>
<td>2:2</td>
<td>4:1</td>
</tr>
<tr>
<td></td>
<td>2:3</td>
<td>3:1</td>
<td></td>
<td>4:2</td>
</tr>
</tbody>
</table>

NextDocumentWithInformation(q, docid)
Similar to NextDocument, but keeps the terms, their positions, frequencies, etc., anything the ranker wants.

\[\text{ProcessQuery}(q, k)\]
1. \(D = \text{PriorityQueue<PairDouble, Integer\\rangle}(k)\);
2. \(d = \text{NextDocumentWithInformation}(q, -1)\);
3. while \(d\).docid \(\neq \infty\) do
4. \(s = \text{ScoreUsingInformation}(d)\);
5. \(D\).push(PairDouble, Integer\\rangle(s, d.docid));
6. if \(|D| > k\) then
7. \(D\).pop();
8. end if
9. \(d = \text{NextDocumentWithInformation}(q, d.docid)\);
10. end while
11. return \(D\);

Term-at-a-time Query Processing
Processing the postings lists one by one, accumulating the scores for the seen documents.

<table>
<thead>
<tr>
<th></th>
<th>salt</th>
<th>water</th>
<th>tropical</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>partial scores</td>
<td>1.1</td>
<td>1.1</td>
<td>1.0</td>
<td>2.2</td>
</tr>
<tr>
<td>old partial scores</td>
<td>1.1</td>
<td>1.1</td>
<td>1.0</td>
<td>2.2</td>
</tr>
<tr>
<td>new partial scores</td>
<td>1.1</td>
<td>1.1</td>
<td>1.0</td>
<td>2.2</td>
</tr>
<tr>
<td>old partial scores</td>
<td>1.1</td>
<td>1.1</td>
<td>1.0</td>
<td>2.2</td>
</tr>
<tr>
<td>tropical</td>
<td>1.1</td>
<td>1.1</td>
<td>1.0</td>
<td>2.2</td>
</tr>
<tr>
<td>final scores</td>
<td>1.1</td>
<td>1.1</td>
<td>1.0</td>
<td>2.2</td>
</tr>
</tbody>
</table>
Term-at-a-time Query Processing

NextDocumentWithInformation(t, q, docid)

- Similar to NextDocumentWithInformation, but process only the postings list of t.

```plaintext
NextDocumentWithInformation(t, q, docid)
```

- Similar to NextDocumentWithInformation, but process only the postings list of t.

```plaintext
PromoteQuery(q, t)
```

- For all i (e, d)
- while d != docid do
- if d != first(docid) then
- d = NextDocumentWithInformation(q, t, d)
- else
- d = first(docid)
- end if
- end while
- return D

Document-at-a-time vs Term-at-a-time

Document-at-a-time is in general more practical.

- Term-at-a-time has the advantage of sequential disk access since it processes one postings list at a time
- Useful in the old days, and in Academic settings
- In commercial search engines, most of the postings lists are cached in memory (in a distributed way), therefore sequential access is inconsequential

Speeding Up Query Processing

Focus on document-at-a-time processing.

- Goal is to return good enough results as fast as possible.
- Good enough == User is happy with the results
- Fast == Processing fewer documents

To reduce the number of documents to be processed, conjunctive processing is assumed, i.e., only documents containing all the query terms are returned.

Exceptions:

- Synonyms
- Stopwords
Simple Strategy: Ignore Documents

Terminating the processing when a predefined number of documents have been processed, even if there are more documents to be processed.

For Web search, this can often be effective for queries on popular contents:
- There are enough relevant pages to satisfy the user’s information needs

However, not ideal for navigational queries:
- There are usually just one or two super relevant pages and the users expect to see them

Simple Strategy: Ignore Postings List

Ignore postings lists that are very costly to process and yet not expected to affect rankings much.

Postings lists for stopwords are usually ignored unless exact matching is required.

Advanced Strategy: Thresholding

Often, the ranking/scoring process is itself quite expensive
- E.g., it might contain expensive machine learning modules

Estimate the score of a document and score it fully only if it is expected to affect the current ranking.

Advanced Strategy: Thresholding

Often, the ranking/scoring process is itself quite expensive
- E.g., it might contain expensive machine learning modules

Estimate the score of a document and score it fully only if it is expected to affect the current ranking.
Advanced Strategy: Reordering

Thresholding becomes more effective if the documents in a postings list are monotonically decreasing in their quality.

Good documents are processed first, and as soon as the system starts seeing documents that are ranked outside of top $k$, the processing can be stopped.

• Instead of ordering the postings according to the docid, we can arrange them in the order of quality, e.g., PageRank.

We won’t be able to compress the postings list as before!

• To compromise, we can bucketize documents on their quality, and within each bucket, order them by docid.

Query Optimization in Real World

Leverages every conceivable technology, from algorithms to hardware.

Very sophisticated and most are trade secrets.

Lecture Review

Review of previous three lectures: Evaluation, Ranking Model, Query Model.

Inverted index data structure and access methods

Encoding and decoding mechanisms for index compression

Index construction algorithms

Query processing approaches and optimization strategies

Next Week: Document Processing.