Data-Intensive Distributed Computing
CS 431/631 451/651 (Winter 2021)

Part 4: Analyzing Text (2/2)

Ali Abedi

These slides are available at https://www.student.cs.uwaterloo.ca/~cs451
This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States
See http://creativecommons.org/licenses/by-nc-sa/3.0/us/ for details
Search!
Abstract IR Architecture

Query Representation

Representation Function

Query Representation

Comparison Function

Index

Document Representation

Representation Function

Indexing

Retrieval

Documents

online

offline
<table>
<thead>
<tr>
<th>Doc 1</th>
<th>Doc 2</th>
<th>Doc 3</th>
<th>Doc 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>one fish, two fish</td>
<td>red fish, blue fish</td>
<td>cat in the hat</td>
<td>green eggs and ham</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Doc 1</th>
<th>Doc 2</th>
<th>Doc 3</th>
<th>Doc 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>1</td>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>cat</td>
<td></td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>egg</td>
<td>1</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>fish</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>green</td>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>ham</td>
<td></td>
<td>1</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>hat</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>one</td>
<td>1</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>red</td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>two</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**postings lists**
(always in sorted order)
<table>
<thead>
<tr>
<th></th>
<th>Doc 1 one fish, two fish</th>
<th>Doc 2 red fish, blue fish</th>
<th>Doc 3 cat in the hat</th>
<th>Doc 4 green eggs and ham</th>
</tr>
</thead>
<tbody>
<tr>
<td>tf</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>blue</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>cat</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>egg</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fish</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>green</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ham</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>hat</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>one</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>red</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>two</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
## Inverted Indexing with MapReduce

<table>
<thead>
<tr>
<th>Doc 1</th>
<th>one fish, two fish</th>
<th>Doc 2</th>
<th>red fish, blue fish</th>
<th>Doc 3</th>
<th>cat in the hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>one</td>
<td>1</td>
<td>red</td>
<td>2</td>
<td>cat</td>
<td>3</td>
</tr>
<tr>
<td>two</td>
<td>1</td>
<td>blue</td>
<td>2</td>
<td>hat</td>
<td>3</td>
</tr>
<tr>
<td>fish</td>
<td>2</td>
<td>fish</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Map**

**Shuffle and Sort:** aggregate values by keys

<table>
<thead>
<tr>
<th>Doc 1</th>
<th>one fish, two fish</th>
<th>Doc 2</th>
<th>red fish, blue fish</th>
<th>Doc 3</th>
<th>cat in the hat</th>
</tr>
</thead>
<tbody>
<tr>
<td>one</td>
<td>1</td>
<td>red</td>
<td>2</td>
<td>cat</td>
<td>3</td>
</tr>
<tr>
<td>two</td>
<td>1</td>
<td>blue</td>
<td>2</td>
<td>hat</td>
<td>3</td>
</tr>
<tr>
<td>fish</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Reduce**

- **cat**: 3 1
- **fish**: 1 2 2 2
- **one**: 1 1
- **red**: 2 1
- **blue**: 2 1
- **hat**: 3 1
- **two**: 1 1
Inverted Indexing: Pseudo-Code

class Mapper {
  def map(docid: Long, doc: String) = {
    val counts = new Map()
    for (term <- tokenize(doc)) {
      counts(term) += 1
    }
    for ((term, tf) <- counts) {
      emit(term, (docid, tf))
    }
  }
}

class Reducer {
  def reduce(term: String, postings: Iterable[(docid, tf)]) = {
    val p = new List()
    for ((docid, tf) <- postings) {
      p.append((docid, tf))
    }
    p.sort()
    emit(term, p)
  }
}
MapReduce sorts the data only based on the key. So if we need the data to be sorted based on a part of the value, we need to move that part to the key.
Inverted Indexing: Pseudo-Code

class Mapper {
    def map(docid: Long, doc: String) = {
        val counts = new Map()
        for (term <- tokenize(doc)) {
            counts(term) += 1
        }
        for ((term, tf) <- counts) {
            emit((term, docid), tf)
        }
    }
}

class Reducer {
    var prev = null
    val postings = new PostingsList()
    def reduce(key: Pair, tf: Iterable[Int]) = {
        if (key.term != prev and prev != null) {
            emit(prev, postings)
            postings.reset()
        }
        postings.append(key.docid, tf.first)
        prev = key.term
    }
    def cleanup() = {
        emit(prev, postings)
    }
}

We still have the memory overflow issue, but the different is that now key.docid is sorted when we add them to the list. As a result, we can compress these values using integer compression techniques to reduce the size of the list.
Postings Encoding

Conceptually:

fish  1  2  9  1  21  3  34  1  35  2  80  3  ...

In Practice:

Don’t encode docids, encode gaps (or $d$-gaps)
But it’s not obvious that this save space...

fish  1  2  8  1  12  3  13  1  1  2  45  3  ...

= delta encoding, delta compression, gap compression
Overview of Integer Compression

Byte-aligned technique
  VarInt (Vbyte)
  Group VarInt

Word-aligned
  Simple family
  Bit packing family (PForDelta, etc.)

Bit-aligned
  Unary codes
  $\gamma/\delta$ codes
  Golomb codes (local Bernoulli model)
VarInt (Vbyte)

Simple idea: use only as many bytes as needed
Need to reserve one bit per byte as the “continuation bit”
Use remaining bits for encoding value

Works okay, easy to implement...

Beware of branch mispredicts!
Simple-9

How many different ways can we divide up 28 bits?

- 28 1-bit numbers
- 14 2-bit numbers
- 9 3-bit numbers
- 7 4-bit numbers

"selectors"

(9 total ways)

Efficient decompression with hard-coded decoders
Simple Family – general idea applies to 64-bit words, etc.
Golomb Codes

\[ x \geq 1, \text{parameter } M: \quad q = \left\lfloor \frac{x - 1}{M} \right\rfloor \quad \text{Encoded in unary} \]

\[ r = x - qM - 1 \quad \text{Encoded in truncated binary} \]

Final result: \((q + 1) r\)

Example:

- \(M = 3, r = 0, 1, 2 (0, 10, 11)\)
- \(M = 6, r = 0, 1, 2, 3, 4, 5 (00, 01, 100, 101, 110, 111)\)
- \(x = 9, M = 3: q = 2, r = 2, \text{code} = 110:11\)
- \(x = 9, M = 6: q = 1, r = 2, \text{code} = 10:100\)

Punch line: optimal \(M \sim 0.69 \frac{N}{df}\)

Different \(M\) for every term!

\[ N = \text{Number of documents} \]
\[ Df = \text{document frequency (the number of documents a term appears in)} \]
We can perform integer compression now!
Chicken and Egg?

But wait! How do we set the Golomb parameter $M$?

Recall: optimal $M \sim 0.69 \left(\frac{N}{df}\right)$

We need the $df$ to set $M$...

But we don’t know the $df$ until we’ve seen all postings!

Write postings *compressed*

Sound familiar?

The problem is that we cannot calculate $df$ until we see all fish *s*
Getting the $df$

In the mapper:
Emit “special” key-value pairs to keep track of $df$

In the reducer:
Make sure “special” key-value pairs come first: process them to determine $df$

Remember: proper partitioning!
**Getting the \textit{df}: Modified Mapper**

<table>
<thead>
<tr>
<th>(key)</th>
<th>(value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fish</td>
<td>1</td>
</tr>
<tr>
<td>one</td>
<td>1</td>
</tr>
<tr>
<td>two</td>
<td>1</td>
</tr>
<tr>
<td>fish</td>
<td>★</td>
</tr>
<tr>
<td>one</td>
<td>★</td>
</tr>
<tr>
<td>two</td>
<td>★</td>
</tr>
</tbody>
</table>

Input document...

Emit normal key-value pairs...

Emit “special” key-value pairs to keep track of \textit{df}...
Getting the $df$: Modified Reducer

First, compute the $df$ by summing contributions from all “special” key-value pair...

Compute $M$ from $df$

Important: properly define sort order to make sure “special” key-value pairs come first!

Write postings compressed

Where have we seen this before?

We have see this before in the pairs implementation of $f(B|A)$ i.e., part 2b
Abstract IR Architecture

- **Query Representation**
- **Document Representation**
- **Comparison Function**
- **Index**
- **Retrieval**
- **Indexing**

**offline**

**online**
MapReduce it?

The indexing problem

- **Perfect for MapReduce!**
- Scalability is critical
- Must be relatively fast, but need not be real time
- Fundamentally a batch operation
- Incremental updates may or may not be important
- For the web, crawling is a challenge in itself

The retrieval problem

- Must have sub-second response time
- For the web, only need relatively few results

Uh... not so good...
Assume everything fits in memory on a single machine...
Boolean Retrieval

Users express queries as a Boolean expression

AND, OR, NOT
Can be arbitrarily nested

Retrieval is based on the notion of sets
Any query divides the collection into two sets: retrieved, not-retrieved
Pure Boolean systems do not define an ordering of the results
Boolean Retrieval

To execute a Boolean query:

1. **Build query syntax tree**
2. **For each clause, look up postings**
3. **Traverse postings and apply Boolean operator**
Term-at-a-Time

Efficiency analysis?
Document-at-a-Time

Tradeoffs?
Efficiency analysis?
Boolean Retrieval

Users express queries as a Boolean expression
AND, OR, NOT
Can be arbitrarily nested

Retrieval is based on the notion of sets
Any query divides the collection into two sets: retrieved, not-retrieved
Pure Boolean systems do not define an ordering of the results

What’s the issue?
Ranked Retrieval

Order documents by how likely they are to be relevant

\[ \text{Estimate relevance}(q, d) \]

Sort documents by relevance
Term Weighting

Term weights consist of two components
Local: how important is the term in this document?
Global: how important is the term in the collection?

Here’s the intuition:
Terms that appear often in a document should get high weights
Terms that appear in many documents should get low weights

How do we capture this mathematically?
Term frequency (local)
Inverse document frequency (global)
TF-IDF* Term Weighting

\[ w_{i,j} = tf_{i,j} \cdot \log \frac{N}{n_i} \]

- \( w_{i,j} \): weight assigned to term \( i \) in document \( j \)
- \( tf_{i,j} \): number of occurrence of term \( i \) in document \( j \)
- \( N \): number of documents in entire collection
- \( n_i \): number of documents with term \( i \)

*Term Frequency-Inverse Document Frequency
Retrieval in a Nutshell

Look up postings lists corresponding to query terms
Traverse postings for each query term
Store partial query-document scores in accumulators
Select top $k$ results to return
Retrieval: Document-at-a-Time
Evaluate documents one at a time (score all query terms)

Accumulators (e.g. min heap)

Document score in top k?
Yes: Insert document score, extract-min if heap too large
No: Do nothing

Tradeoffs:
Small memory footprint (good)
Skipping possible to avoid reading all postings (good)
More seeks and irregular data accesses (bad)
Retrieval: Term-At-A-Time
Evaluate documents one query term at a time
Usually, starting from most rare term (often with $tf$-sorted postings)

Tradeoffs:
Early termination heuristics (good)
Large memory footprint (bad), but filtering heuristics possible