INVERTED INDEX CONSTRUCTION
Outline

- Intro
- Basics of probability and information theory
- Retrieval models
- Retrieval evaluation
- Link analysis

- **From queries to top-k results**
  - Query processing
  - Index construction
  - Top-k search

- Social search
Overview

This is a sample document for explanation purposes.

Queue of crawled documents

This is a sample document for explanation purposes.

Tokenization

Linguistic processing

Thesaurus

sample document explanation purpose

Indexing

B+ tree

Inverted index

document ... explanation ... purpose ... sample

15: 0.03
43: 0.025
51: 0.015
53: 0.08
55: 0.061
...
11: 0.02
16: 0.033
43: 0.015
54: 0.021
...
17: 0.011
43: 0.045
58: 0.015
...
9: 0.03
12: 0.04
21: 0.015
43: 0.02
...
Inverted index from the term-document matrix

- How to store a realistic term-document matrix with millions of terms and hundreds of millions of documents?

- Obviously a document contains relatively few terms.
  → Document vectors contain many zeros.
  → The whole matrix contains a lot more zeros than ones.

- Store for each term only the IDs of the documents in which it occurs, along with scores.

<table>
<thead>
<tr>
<th>Term</th>
<th>$d_1$</th>
<th>$d_2$</th>
<th>$d_3$</th>
<th>$d_4$</th>
<th>$d_5$</th>
<th>$d_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>champion</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>football</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>goal</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>law</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>party</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>politician</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rain</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>score</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>soccer</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>weather</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>wind</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Important steps for index constructions

- Sort documents by terms.

- Merge multiple term occurrences in a single document but maintain position information and add frequency information.

- Construct corpus vocabulary with entries of the form (term, #docs, corpus_count)

- Construct for every term postings with entries of the form (docID, count, list[pos1, offsets..])

  Why are position-based postings better than postings that store biwords or longer phrases (e.g., ‘stanford university’ or ‘hasso plattner institute’)?

- All steps involve distributed computations (e.g., through MapReduce methods)
### Example

#### Vocabulary

<table>
<thead>
<tr>
<th>term</th>
<th>#docs</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>champion</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>football</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>goal</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>law</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>party</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>politician</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>rain</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>score</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>soccer</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weather</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>wind</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

#### Frequency-based postings (offsets omitted)

<table>
<thead>
<tr>
<th>docID</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Pointers

<table>
<thead>
<tr>
<th>term</th>
<th>d_1</th>
<th>d_2</th>
<th>d_3</th>
<th>d_4</th>
<th>d_5</th>
<th>d_6</th>
</tr>
</thead>
<tbody>
<tr>
<td>champion</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>football</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>goal</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>law</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>party</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>politician</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rain</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>score</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>soccer</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>weather</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>wind</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Distributed index construction with MapReduce

- Programming paradigm for scalable, highly parallel data analytics
- Scheduling, load balancing and fault tolerance are core ingredients
- Enables distributed computations on 1000’s of machines
- Programming based on key-value pairs:

  \[ \text{Map: } K \times V \rightarrow (L \times W)^* \]
  \[ (k, v) \mapsto (l_1, w_1), (l_2, w_2), ... \]

  \[ \text{Reduce: } L \times W^* \rightarrow W^* \]
  \[ l, (x_1, x_2, ...) \mapsto y_1, y_2, ... \]
Possible MapReduce Infrastructure for Indexing

- MapReduce implementations: PIG (Yahoo), Hadoop (Apache), DryadLinq (Microsoft), Facebook Corona

Source: Introduction to Information Retrieval
TF computation with MapReduce

- **Step 1**
  - **Map:**
    \[(\text{docID}, \text{content}) \rightarrow \{((\text{term}, \text{docID}), 1), \ldots\}\]
  - **Reduce:**
    \[((\text{term}, \text{docID}), \{1,\ldots\}) \rightarrow \{((\text{term}, \text{docID}), \text{count})\}\]

- **Step 2**
  - **Map:**
    \[((\text{term}, \text{docID}), \text{count}) \rightarrow \{(\text{docID}, (\text{term}, \text{count}))\}\]
  - **Reduce:**
    \[(\text{docID}, \{(\text{term}, \text{count}), \ldots\}) \rightarrow \{(\text{docID}, \text{term}), (\text{count/doc\_length}), \ldots\}\]
Example (1)

Computing term counts: Map

- example document
- simple example
- computation term frequency in document
- document, 1
- example, 1
- example, 1
- simple, 1
- computation, 1
- document, 1
- frequency, 1
- term, 1
Example (2)

- Computing term counts: **Reduce**

```
document, 1
document, {1,1}
document, 2
example, 1
example, {1, 1}
example, 2
example, {1}
simple, {1}
simple, 1
computation, {1}
computation, 1
computation, 2
frequency, {1}
frequency, 1
frequency, 2
term, {1}
term, 1
term, 2
```
30 billion documents

On avg. term occurs in 
~ 100 documents

10 Mio. distinct terms

~ 3 × 10^{12} entries for the postings
10 Mio. entries for the vocabulary

Assume ~5 Bytes per entry
~ 15 TB in total

Question:
How are the vocabulary and the postings stored?
Storing the vocabulary: B+ trees

- Balanced search tree over the key space with high node fanout

![Diagram of B+ tree structure with nodes and keys representing the vocabulary storage.](image)
Properties of B+ trees

- Every B+ tree is balanced

- Ordered partitioning of the key space

- In a B+ tree of order $n$ (i.e., with fanout size $n$) every internal node, except the root, has $m$ children, with $\lfloor n/2 \rfloor \leq m \leq n$

- For the root: $2 \leq m \leq n$

- For the leaf nodes: $\lfloor n/2 \rfloor \leq m \leq n - 1$

→ How could the insertion, deletion of keys be done?
Properties of B+ trees

- The maximum number of entries stored in a B+ tree of order $n$ and height $h$ is $n^h - n^{h-1}$
  → a 4-level B+ tree of order $n = 100$ would be sufficient to store 10 Mio. term keys

- The minimum number of entries stored in a B+ tree of order $n$ and height $h$ is $2 \left\lceil \frac{n}{2} \right\rceil^{h-1}$

- Space required: $O(|K|)$, where $K$ is the set of keys
- Insertion, deletion, finding: $O(\log_n(|K|))$

- Typically, the upper levels (up to the leaf level) of the B+ tree are loaded in main memory, the information linked with the leaves resides on disk.
B+ tree construction through bulk-loading

- Sort the entries by key values.

- Start with empty page as root node and insert a pointer to the first page of entries.

- Continue with the next page, insert its smallest key value into the root as separation key and insert pointer to this page. Repeat this step until the root is full.

- When the root is full, split it and create a new root.

- Keep inserting entries into the right most index node above the leaves, split the node when it is full and continue recursively.
Index merging

Source: Modern Information Retrieval
Dynamic Index

- On the web, pages are constantly added, deleted, modified

- Solution
  - Use index $I_0$ for the static pages
  - Use index $I_+$ for documents that are added
  - Use index $I_\sim$ for documents that are frequently modified
  - Use index $I_-$ for documents that are deleted

- Complete index: $(I_0 \cup I_+ \cup I_\sim) \setminus I_-$
Final Index

B+ tree (or other search tree on vocabulary)

(document, $idf_1$) ... (explanation, $idf_2$) ... (purpose, $idf_3$) ...

15: 0.03
43: 0.025
51: 0.015
53: 0.018
55: 0.061

11: 0.02
16: 0.033
43: 0.015
54: 0.021

17: 0.011
43: 0.045
58: 0.015

How to store the vocabulary efficiently?

Inverted lists (posting lists) ... may contain hundreds of thousands of entries

Term IDs

Vocabulary terms
Vocabulary compression (1)

- With naive dictionary storage:

<table>
<thead>
<tr>
<th>term</th>
<th>document frequency</th>
<th>pointer to postings list</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>656,265</td>
<td>→</td>
</tr>
<tr>
<td>aachen</td>
<td>65</td>
<td>→</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>zulu</td>
<td>221</td>
<td>→</td>
</tr>
</tbody>
</table>

20 bytes | 4 bytes | 4 bytes

Source: Introduction to Information Retrieval

- In Unicode: \((2 \times 20 + 4 + 4)\) bytes per term
- For 10 Mio. terms: \(~ 460\) MB needed
  → fixed-width entries too wasteful
Better strategy: Vocabulary as sequence of terms

... much more space-efficient than previous scheme

<table>
<thead>
<tr>
<th>freq.</th>
<th>postings ptr.</th>
<th>term ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

4 bytes 4 bytes 3 bytes

Source: Introduction to Information Retrieval

Pointers mark the beginning and the end of a vocabulary term.
Vocabulary compression (3)

- Save more space by
  - Grouping k subsequent terms (k-1 pointers are saved per group)
  - Prefix replacement

One block in blocked compression (k = 4) …
8automata8automate9automatic10automation

… further compressed with front coding.
8automat*a1*e2*ic3*ion

Source: Introduction to Information Retrieval
Comparison of vocabulary compression strategies

- Compression of vocabulary with ~400,000 terms:

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Size in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>dictionary, fixed-width</td>
<td>11.2</td>
</tr>
<tr>
<td>dictionary, term pointers into string</td>
<td>7.6</td>
</tr>
<tr>
<td>~, with blocking, $k = 4$</td>
<td>7.1</td>
</tr>
<tr>
<td>~, with blocking &amp; front coding</td>
<td>5.9</td>
</tr>
</tbody>
</table>

Source: [Introduction to Information Retrieval](http://www.example.com)
For vocabularies of moderate size (e.g., for in-memory processable size) use tries (conceptually the same as the previous scheme)

This is a text.

This text has many letters.

Terms are made of letters.

letters: (d2, 1, [5]), (d3, 1, [5])

made: (d3, 1, [3])

many: (d2, 1, [4])

text: (d1, 1, [4]), (d2, 1, [2])

terms: (d3, 1, [1])
Tries vs. hash tables

B+ tree (or other search tree on vocabulary)

(document, $idf_1$) ... (explanation, $idf_2$) ... (purpose, $idf_3$) ...

15: 0.03
43: 0.025
51: 0.015
53: 0.018
55: 0.061
...

11: 0.02
16: 0.033
43: 0.015
54: 0.021
...

17: 0.011
43: 0.045
58: 0.015
...

How are the inverted lists stored?

Inverted lists (posting lists) ... may contain hundreds of thousands of entries

Term IDs

Terms

Dr. Gjergji Kasneci | Introduction to Information Retrieval | WS 2012-13
Storing inverted lists

- Partition the list in blocks of same size

- Blocks are stored sequentially
  - We will see later that for Boolean queries sorting by ID is sufficient, for ranking sorting by scores (i.e., term frequencies) is better

- Skip pointers at the beginning of each block point either to the next block or a few blocks ahead

11 (…)
17 (…)
23 (…)
27 (…)
59 (…)
71 (…)
73 (…)
90 (…)
103 (…)
Compressing inverted lists

- Given a Zipf-distribution of terms over the indexed documents, the lengths of the inverted lists will follow the same distribution.
  - Unbalanced latencies for reading lists of highly varying sizes from disk

- Is it possible to mitigate these latencies?
  - Effective compression needed

- Could we apply Ziv-Lempel compression to inverted list entries?

- Ziv-Lempel is good for continuous text but not for postings

- For inverted lists, gaps between successive doc IDs are encoded
Unary encoding of gaps

- Gap size $k$ is encoded by $(k - 1)$-times 0 followed by one 1

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Unary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>01</td>
</tr>
<tr>
<td>3</td>
<td>001</td>
</tr>
<tr>
<td>4</td>
<td>0001</td>
</tr>
<tr>
<td>5</td>
<td>00001</td>
</tr>
<tr>
<td>6</td>
<td>000001</td>
</tr>
<tr>
<td>7</td>
<td>0000001</td>
</tr>
<tr>
<td>8</td>
<td>00000001</td>
</tr>
<tr>
<td>9</td>
<td>000000001</td>
</tr>
<tr>
<td>10</td>
<td>0000000001</td>
</tr>
</tbody>
</table>

- Optimal for $P(\Delta = k) = \left(\frac{1}{2}\right)^k$
Gap size $k$ is encoded by its binary representation

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Unary</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>001</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>0001</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>00001</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>000001</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>0000001</td>
<td>110</td>
</tr>
<tr>
<td>8</td>
<td>00000001</td>
<td>111</td>
</tr>
<tr>
<td>9</td>
<td>000000001</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>0000000001</td>
<td>1010</td>
</tr>
</tbody>
</table>

Good for long gaps (but not prefix-free)
Elias Gamma encoding of gaps

- Gap size $k$ is encoded by $1 + \lfloor \log_2 k \rfloor$ in unary followed by binary representation, without the most significant bit.
- E.g.: $9 \rightarrow 0001\ 001$

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Unary</th>
<th>Binary</th>
<th>Gamma</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>01</td>
<td>10</td>
<td>01 0</td>
</tr>
<tr>
<td>3</td>
<td>001</td>
<td>011</td>
<td>01 1</td>
</tr>
<tr>
<td>4</td>
<td>0001</td>
<td>100</td>
<td>001 00</td>
</tr>
<tr>
<td>5</td>
<td>00001</td>
<td>101</td>
<td>001 01</td>
</tr>
<tr>
<td>6</td>
<td>000001</td>
<td>110</td>
<td>001 10</td>
</tr>
<tr>
<td>7</td>
<td>0000001</td>
<td>111</td>
<td>001 11</td>
</tr>
<tr>
<td>8</td>
<td>00000001</td>
<td>1000</td>
<td>0001 000</td>
</tr>
<tr>
<td>9</td>
<td>000000001</td>
<td>1001</td>
<td>0001 001</td>
</tr>
<tr>
<td>10</td>
<td>0000000001</td>
<td>1010</td>
<td>0001 010</td>
</tr>
</tbody>
</table>

- Optimal for $P(\Delta = k) \approx \frac{1}{2k^2}$
Google’s Gamma encoding scheme

Source: [WSDM 2009 keynote by J. Dean](http://example.com/WSDM2009_keynote)

Block format (with $N$ documents and $H$ hits):

- Delta to last docid in block: varint
- Block length: varint
- Encoding type: Gamma
- # docs in block: Gamma
- $N - 1$ docid deltas: Rice$_k$ coded
- $N$ values of # hits per doc: Gamma
- $H$ hit attributes: run length Huffman encoded
- $H$ hit positions: Huffman-Int encoded
Other types of indeces

- **Suffix trees**

- Index for regular expression queries (e.g. \textit{Permuterm Index} for wildcard queries)

- **R+ trees** for spatial data

- Index with temporal information (for temporal queries)

- ...

Dr. Gjergji Kasneci | Introduction to Information Retrieval | WS 2012-13
Summary

- Steps to index construction
  - Sorting docs by terms
    - vocabulary construction
    - postings construction
  (Parallelization through MapReduce)

- Making the vocabulary efficiently searchable with B+ trees
  - Vocabulary compression (sequential term storage with blocking and prefix replacement)

- Prefix trees for maintaining vocabulary of moderate size in main memory

- Storing and compressing inverted lists
  - Equal-size blocks with pointers between subsequent blocks
  - Gap-based encoding within blocks (Unary, Gamma, Rice, ...)