Indexing and Querying

Avishek Anand
Inverted Index Construction and Query Processing

- Inverted Indexing basics revisited
- Indexing Static Collections
  - Dictionaries and Forward Index
  - Inverted Index Organisation
  - Scalable Indexing
- Query Processing
  - DAAT vs TAAT
  - Weak AND
Why do we index text collections?

How do we index documents?

What are the data structures?

What are the design decisions for organising the index?

How do we index huge collections?

How do we process queries efficiently over postings list?
Why do we index text collections?
- Efficient document retrieval

How do we index documents?
- What are the data structures?
  - lexicon, inverted lists
- What are the design decisions for organising the index?
  - document order, score order
- How do we index huge collections?
  - distributed indexing, term/doc partitioning
- How do we process queries efficiently over postings list?
  - query processing strategies
Terminology Recap

Terms, documents, collection

Lexicon

Information retrieval

Stemming, stop-word rem

Queries, results

Index, lexicon, posting, posting list
Lexicon or Dictionary

- Maintains statistics and information about the indexed unit (word, n-gram etc)

  `< hannover ; location: 82271 ; tid:12 ; df:23, ... >`

- Posting list location - for posting list retrieval

- Term identifier - for term lookups, matching and range queries

- Document frequency and associated statistics - for ranking

- Data Structures for Lexicon

  - Hash-based Lexicon

  - B+-Tree based Lexicon
Figure 4.2 Dictionary data structure based on a hash table with $2^{10} = 1024$ entries (data extract from schema-independent index for TREC45). Terms with the same hash value are arranged in a linked list (chaining). Each term descriptor contains the term itself, the position of the term’s postings list, and a pointer to the next entry in the linked list.

In GOV2 is 9.2 bytes. Storing each term in a fixed-size memory region of 20 bytes wastes 10.8 bytes per term on average (internal fragmentation).

One way to eliminate the internal fragmentation is to not store the index terms themselves in the array, but only pointers to them. For example, the search engine could maintain a primary dictionary array, containing 32-bit pointers into a secondary array. The secondary array then contains the actual dictionary entries, consisting of the terms themselves and the corresponding pointers into the postings file. This way of organizing the search engine’s dictionary data is shown in Figure 4.3. It is sometimes referred to as the dictionary-as-a-string approach, because there are no explicit delimiters between two consecutive dictionary entries; the secondary array can be thought of as a long, uninterrupted string.

For the GOV2 collection, the dictionary-as-a-string approach, compared to the dictionary layout shown in Figure 4.1, reduces the dictionary’s storage requirement by $10^8 - 4 = 6.8$ bytes per entry. Here the term 4 stems from the pointer overhead in the primary array; the term 10.8 corresponds to the complete elimination of internal fragmentation.

It is worth pointing out that the term strings stored in the secondary array do not require an explicit termination symbol (e.g., the "\0" character), because the length of each term in the dictionary is implicitly given by the pointers in the primary array. For example, by looking at the pointers for "shakespeare" and "shakespearean" in Figure 4.3, we know that the dictionary entry for "shakespeare" requires $16629970 - 16629951 = 19$ bytes in total: 11 bytes for the term plus 8 bytes for the 64-bit file pointer into the postings file.
Hash-Based Lexicon

- Constant lookups based on a Hash table
- Entire Lexicon loaded to the memory
Hash-Based Lexicon

- Constant lookups based on a Hash table

- Entire Lexicon loaded to the memory

- Updates difficult

- Range Searches, Matching, Substring queries not supported
**B+-Tree or Sort-based Lexicon**

- **B+-Tree**: Leaf nodes additionally linked for efficient range search
- Supports lookups in $O(\log n)$ and range searches in $O(\log n + k)$
- Vocabulary dynamics (i.e., new or removed terms) no problem
- Works on *secondary storage*
• Mapping of doc-ids to term-ids in the same order

1: “what does the fox say?”

1: 124 53 1 49935 100

• Efficient retrieval of terms from (already parsed) text
• snippet generation
• proximity features for proximity-aware ranking
• per-doc term distribution for query expansions
Inverted index is a collection of posting lists

Posting contains document identifiers (as integers) along with scores (integers or doubles) and possibly positions (as integers)

Postings list can be organised according to

- document identifiers - document ordering
- scores - Impact ordering

What are the merits of these orderings?
Document Order vs Score Order

**Document Ordering**
- Based on faster intersections
- High compression of index using gap encoding of dids
- Easily updatable

**Score/Impact Ordering**
- Based on processing Top-k results fast
- Low compression ratio
- Difficult to update

Index organisation depends on query processing style.
Inverted Index Construction

- We are given a set of documents $D$, where each document $d$ is considered as a bag of terms

- Inverted Lists are created by a process termed as Inversion

- **Memory-based** Inversion
  - Takes place entirely in-memory
  - For small collections, where the index + lexicon fits in memory

- **Disk-based** Inversion
  - Sort-based inversion vs Merge-based inversion
A dictionary is required that allows efficient single-term lookup and insertion operations.

An extensible (i.e., dynamic) list data structure is needed that is used to store the postings for each term.

1: “what does the fox say?”

2: “the fox jumped over the fence”

doc: [term, positions]

1: [the, <1,5>]

[term, posting list]

“the”: [1, <3>] [2, <1,5>]
Sort-based Inversion

- Input Collection D >> memory size M

- Inversion can be seen as a sort operation on the term identifiers

- This method is based on external sort over data which does not fit into the memory
  - Read data of size M into memory, sort them and write back to disk
  - Multiway merge of D/M sorted lists to create index

- Shortcomings
  - Dictionary might not fit in-memory
  - Large memory requirements due to intermediate data
Exercise 1: Analysis of Sort-based Inversion

Simple Computational Model

- Total number of postings = N
- Number of postings which fit in memory = M
- Cost of disk read/write of a posting = c

- What is the estimated cost of sort-based Inversion in terms of N, M, and c?

- How does the cost compare with in-memory sort-based inversion (assuming we had enough memory or N > M)?
Merge-based Inversion

- Generalisation of in-memory indexing
- Reads input collection to create an in-memory index of size $M$ and write it to disk to create partial indexes with local lexicons
- Compression in posting lists in partial indexes
- Multiway Merge of corresponding lists from the partial indexes to create one consolidated index
Map-Reduce crash course

- Programming paradigm for distributed data processing
- Improves overall throughput by parallelising loading of data
- Data is partitioned into the nodes which process the data in the following phases
  - **Map**: Generates (key, value) pairs
  - **Shuffle**: Shuffles the pairs over the network to the reducers
  - **Reduce**: operates on all values for the same key are
Map-Reduce Example: Word Count

1: “what does the fox say ?”

Mapper - 1
what : 1
does : 1
the : 1
fox : 1
say : 1

Mapper - 2
jumped : 1
over : 1
the : 2
fox : 1
fence : 1

Shuffle + Sort

Reducers aggregate freq.

Reducer - 1
does : 1
fence : 1
jumped : 1
fox : 1
over : 1
the : 1
say : 1
what : 1

Reducer - 2
+ fox : 1
+ the : 2
Exercise 2: Index Construction using Map-Reduce

• How would you build the inverted index using Map-reduce?

• What are the key-value pairs as defined by the Mapper?

• What does the reducer do with the values of the same key?
Query Semantics

- Boolean Queries — keywords form a boolean expression
  
  - Conjunctive, Disjunctive, Negation, combination
  
  - (“spears” AND “britney”) OR (“lady” AND “gaga”)

- Phrase queries — entire phrase present in the same order
  
  - “to be or not be that is the question”

- Advanced Queries

  - Temporal queries — “summer olympics @ [2001 - 2003]”

  - Proximity aware queries — keywords should be present within a gap of each other

  - Wild card queries — “a * saved is a * earned”
Document Order vs Score Order

Document Ordering

- Based on faster intersections
- High compression of index using gap encoding of dids
- Easily updatable

Score/Impact Ordering

- Based on processing Top-k results fast
- Low compression ratio
- Difficult to update

Index organisation depends on query processing style.
• Best choice for boolean queries are document ordered lists

• Postings are stored in a column major format

  • A list of all document identifiers — good for compression and faster boolean operations

• A list of all scores

• Scores are typically aggregated by weighted sum of the partial term scores
Term-At-A-Time

hannover

12  23  48  71  93  96  101

messe

18  23  71  77  112  189

Term-at-a-time

- Process one list at a time
- Maintain accumulators for partial results and update them
- Best for unions

Accumulators in memory
Term-At-A-Time

- Process one list at a time
- Maintain accumulators for partial results and update them
- Best for unions

hannover

|   | 12 | 23 | 48 | 71 | 93 | 96 | 101 |

messe

|   | 18 | 23 | 71 | 77 | 112 | 189 |

Accumulators in memory

<table>
<thead>
<tr>
<th></th>
<th>12</th>
<th>71</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>23</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td></td>
</tr>
</tbody>
</table>
Term-At-A-Time

hannover

12 23 48 71 93 96 101

gesse

18 23 71 77 112 189

Term-at-a-time

✦ Process one list at a time
✦ Maintain accumulators for partial results and update them
✦ Best for unions

Accumulators in memory

18 12 71
23 77 93 96
48 101 112
189
Document-At-A-Time

**Document-at-a-time**
- Open cursors to all lists
- Systematically move cursors to satisfy boolean expression
- Best for intersections

**Conjunctive query semantics**
- In each iteration find the max did M
- Move other cursors to greater or equal to M
- If all cursors point to M, move all one step further
Document-At-A-Time

- Open cursors to all lists
- Systematically move cursors to satisfy boolean expression
- Best for intersections

Conjunctive query semantics
- In each iteration find the max did M
- Move other cursors to greater or equal to M
- If all cursors point to M, move all one step further
Document-At-A-Time

- Open cursors to all lists
- Systematically move cursors to satisfy boolean expression
- Best for intersections

hannover
- 12 23 48 71 93 96 101

messe
- 18 23 71 77 112 189

 Conjunctive query semantics
- In each iteration find the max did M
- Move other cursors to greater or equal to M
- If all cursors point to M, move all one step further
Skip Lists

What if we are allowed to store extra information for each lists?
Skip Lists

What if we are allowed to store extra information for each lists?

- Skip list allow fast intersections acting as a secondary index over posting lists
- Typically skip over fixed number of postings and are square root of the length of the postings list
Skip Lists

What if we are allowed to store extra information for each lists?

- Skip list allow fast intersections acting as a secondary index over posting lists
- Typically skip over fixed number of postings and are square root of the length of the postings list
Simple Computational Model

Average length of a posting list = n
Average number of terms in the query = m

1. What is the complexity (space and time) of both methods?

2. What method would you use for phrase queries e.g. “intel inside”?

3. How would you use the positional information for phrase queries?**
http://www.ir.uwaterloo.ca/book/


Further further Reading

- C. Buckley, A. F. Lewit: Optimization of Inverted Vector Searches. SIGIR 1985
- Martin Theobald, Ralf Schenkel, Gerhard Weikum: Efficient and self-tuning incremental query expansion for top-k query processing. SIGIR 2005