Indexing and Querying

Avishek Anand
Inverted Index Construction and Maintenance

- Inverted Indexing basics revisited
- Indexing Static Collections
  - Dictionaries
  - Forward Index
  - Inverted Index Organisation
- Scalable Indexing
- Indexing Dynamic Collections
Query Processing over Inverted Indexes

- Query Processing over Document-id ordered lists
  - Document-at-a-Time vs Term-at-a-Time Processing
  - WAND Processing

- Top-K Processing
  - Fagin’s top-k
  - TA, NRA and CA

- Supporting advanced queries - phrases, proximity-aware, temporal queries
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 index evolving or dynamic collections?
Text Collections and Indexing

✦ 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 index evolving or dynamic collections?
   index maintenance strategies
Terminology Recap

- terms, documents, collection
- stemming, stop-word removal
- information retrieval
- lexicon
- 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
Dictionary data structure based on a hash table with $2^{10} = 1024$ entries (data extracted 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 $10^8 / 4 = 6$ bytes per entry. Here the term 4 stems from the pointer overhead in the primary array; the term 10 correponds to the complete elimination of an 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.
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

Figure 4.2
Dictionary data structure based on a hash table with $2^{10} = 1024$ entries (data extracted 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 corresponds to the complete elimination of any 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.
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**
Forward Index

- Mapping of doc-ids to term-ids in the same order

1: “what does the fox say?”

<table>
<thead>
<tr>
<th></th>
<th>124</th>
<th>53</th>
<th>1</th>
<th>49935</th>
<th>100</th>
</tr>
</thead>
</table>

- 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

- 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 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”
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
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
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

Mappers emit <word, freq>

Shuffle + Sort

2: “the fox jumped over the fence”

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

**Reducer - 2**
- what : 1
- the : 2

Reducers aggregate freq.
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?
How do we deal with dynamically growing collections?

- Real world document collections are often dynamic
- Index Maintenance: How do we keep the index consistent with the changes or updates to the document collection?
- Challenge of Time: Not enough time to rebuild indexes
- Challenge of query competitiveness: Queries to be served in reasonable time
How do we deal with dynamically growing collections?

- Real world document collections are often dynamic.

- Index Maintenance: How do we keep the index consistent with the changes or updates to the document collection?

- Challenge of Time: Not enough time to rebuild indexes.

- Challenge of query competitiveness: Queries to be served in reasonable time.
How do we deal with dynamically growing collections?

- Real world document collections are often dynamic
- Index Maintenance: How do we keep the index consistent with the changes or updates to the document collection?
- Challenge of Time: Not enough time to rebuild indexes
- Challenge of query competitiveness: Queries to be served in reasonable time
Indexing Dynamic Collections

How do we deal with dynamically growing collections?

- Real world document collections are often dynamic
- Index Maintenance: How do we keep the index consistent with the changes or updates to the document collection?
- Challenge of Time: Not enough time to rebuild indexes
- Challenge of query competitiveness: Queries to be served in reasonable time
How do we deal with dynamically growing collections?

- Real world document collections are often dynamic
- Index Maintenance: How do we keep the index consistent with the changes or updates to the document collection?
- Challenge of Time: Not enough time to rebuild indexes
- Challenge of query competitiveness: Queries to be served in reasonable time
How do we deal with dynamically growing collections?

- Real world document collections are often dynamic
- Index Maintenance: How do we keep the index consistent with the changes or updates to the document collection?
- Challenge of Time: Not enough time to rebuild indexes
- Challenge of query competitiveness: Queries to be served in reasonable time
Index Maintenance

- **Single Index**: Maintain one index for the entire document collection by recomputing whenever there are updates (maybe batch updates)
  - Efficient query processing but high maintenance cost

- **Multiple Partial Indexes**: No index re-computation. Partial index finalized once memory is full.
  - Query is processed over each partial index and results are merged hence queries are slower
Index Maintenance

- **Single Index**: Maintain one index for the entire document collection by recomputing whenever there are updates (maybe batch updates)
  - Efficient query processing but high maintenance cost

- **Multiple Partial Indexes**: No index re-computation. Partial index finalized once memory is full.
  - Query is processed over each partial index and results are merged hence queries are slower
Merging Indexes

- Merging of two posting lists
  - In-place - keep free space after index blocks for updates (more space)
  - Chained-merge/merge-based— chain updates in a new block (slower access)
  - Merging is possible since compression techniques are local
Merging Indexes

- Merging of two posting lists
  - In-place - keep free space after index blocks for updates (more space)
  - Chained-merge/merge-based— chain updates in a new block (slower access)
- Merging is possible since compression techniques are local
Merging Indexes

- Merging of two posting lists

  - In-place - keep free space after index blocks for updates (more space)
  - Chained-merge/merge-based— chain updates in a new block (slower access)

- Merging is possible since compression techniques are local
Merging Indexes

- Merging of two posting lists
  - In-place - keep free space after index blocks for updates (more space)
  - Chained-merge/merge-based— chain updates in a new block (slower access)
- Merging is possible since compression techniques are local
Merging Indexes

- Merging of two posting lists
  - In-place - keep free space after index blocks for updates (more space)
  - Chained-merge/merge-based— chain updates in a new block (slower access)
- Merging is possible since compression techniques are local
Merging Indexes

- Merging of two posting lists
  - In-place - keep free space after index blocks for updates (more space)
  - Chained-merge/merge-based— chain updates in a new block (slower access)
- Merging is possible since compression techniques are local
Each on-disk index has generation number $g$
- In-memory index has $g = 0$
- When two on-disk indexes have same $g = i$, they are merged to form on-disk index having $g = i+1$.

Logarithmic merge results in $\log_2 N$ partitions
- $N$ is the number of in-memory blocks generated

Generalized Logarithmic merge : $\log_k N$ partitions
- Also called as Lazy merge or $k$-constraint logarithmic merge
Logarithmic Merge

Lazy Merge

In-memory Index

Timeline

[Buttcher et al. SIGIR '06]
Geometric Merge

• Each partition contains an inverted index
• Index sizes form a geometric series with ratio $r$

$$r = \frac{\text{Size of index at partition } K+1}{\text{Size of index at partition } K}$$

• a partition $k$ has index of size 0 or $[r^{k-1} M, (r-1)r^{k-1} M]$ postings
• Increasing the value of $r$, increases the number of merges thus reducing the number of partitions
  - Immediate merge is a geometric merge with $r = \infty$
Geometric Merge

Geometric Partitioning

Active Merge

In-memory Index

Timeline

r = 3

r = 3 4 5
Exercise -3 : Analysis of Merging Techniques

Simple Computational Model

Total number of postings = N
Number of postings which fit in memory = M
Cost of sequential access = c, random access = 1000c

1. What is the cost estimate (in terms of disk operations involved) for immediate merge?

2. Compare Query processing estimates in terms of c for all merge methods.

3. What is the cost estimate for geometric or logarithmic merge?***
References

http://www.ir.uwaterloo.ca/book/


## Index Construction - Computational Model

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Symbol</th>
<th>Assumed Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total text size</td>
<td>$B$</td>
<td>$5 \times 10^9$ bytes</td>
</tr>
<tr>
<td>Number of docs</td>
<td>$N$</td>
<td>$5 \times 10^6$</td>
</tr>
<tr>
<td>Number of distinct words</td>
<td>$n$</td>
<td>$1 \times 10^6$</td>
</tr>
<tr>
<td>Total number of words</td>
<td>$F$</td>
<td>$800 \times 10^6$</td>
</tr>
<tr>
<td>Number of index pointers</td>
<td>$f$</td>
<td>$400 \times 10^6$</td>
</tr>
<tr>
<td>Final size of compressed inv. file</td>
<td>$I$</td>
<td>$400 \times 10^6$ bytes</td>
</tr>
<tr>
<td>Disk seek time</td>
<td>$t_s$</td>
<td>$10 \times 10^{-3}$ sec</td>
</tr>
<tr>
<td>Disk transfer time per byte</td>
<td>$t_r$</td>
<td>$0.5 \times 10^{-6}$ sec</td>
</tr>
<tr>
<td>Inverted file coding per byte</td>
<td>$t_d$</td>
<td>$5 \times 10^{-6}$ sec</td>
</tr>
<tr>
<td>Time to compare and swap 10-byte records</td>
<td>$t_c$</td>
<td>$10^{-6}$ sec</td>
</tr>
<tr>
<td>Time to parse, stem and look up one term</td>
<td>$t_p$</td>
<td>$20 \times 10^{-6}$ sec</td>
</tr>
<tr>
<td>Amount of main memory available</td>
<td>$M$</td>
<td>$40 \times 10^6$ bytes</td>
</tr>
</tbody>
</table>