Indexing

(COSC 488)
Nazli Goharian
nazli@cs.georgetown.edu

Efficiency

• Difficult to analyze sequential IR algorithms: data and query dependency (query selectivity).
• $O(q(\text{cf}_{\text{max}}))$ -- high estimate
• No standard analytical model to estimate query performance, hence empirical efforts.
Efficiency Techniques

- Indexing
  - Compression
- Index Pruning (Top Doc)
- Efficient Query Processing
- Duplicate Document Detection

Indexing

- Scanning Text
  - Small document collection
- Inverted index [1960’s]
  - Reducing I/O, thus, speeding query processing; storage overhead; time overhead to build index
- Signature files
  - Smaller and faster; less functionality
- Relational
  - Higher overhead; supports integration of structured data and text
Inverted Index

• Regardless of the retrieval strategy we need a data structure to efficiently store:
  – For each term in the document collection
    • The list of documents that contain the term
    • Number of documents having a term (\(df, idf\))
    • For each occurrence of a term in a document
      – The frequency the term appears in the document (\(tf\))
      – The position in the document for which the term appears (only needed if proximity search is supported).
        » Position may be expressed as section, paragraph, sentence, location within sentence.

Inverted Index

• Associates a posting list with each term

  a: \((D1,7) (D2,5) (D3,19) (D4,11)\)…
  abacus: \((D7,1)\)
  abatement: \((D15,1) (D23,2)\)
  …
  zoology: \((D8,1) (D32,2)\)

• Inverted because it lists for a term, all documents that contain the term.
Inverted Index: Structure

- **Document map** (Document information: url, length, page rank, …)
- **Term list/index** (Lexicon/Vocabulary/Dictionary) - stores distinct terms and document frequency information (df, idf)
- **Posting list** - stores documents for a given term

![Diagram of Inverted Index Structure]

Skip Pointers

- To optimize
  - Join operation of O(m+n) for posting lists of size $m$ and $n$
  - Search for a given document $d$ in the PL (*will be discussed later*)

![Diagram of Skip Pointers]

Q: $<t1 \text{ AND } t2>$
Query Processing using Inverted Index

• Term-at-a-time:
  – For each term, at a time, the inverted index is accessed to calculate scores

• Document-at-a-time:
  – All inverted lists (posting lists relevant to the query) are accessed concurrently. In case of intersections between PLs, forward-skip optimizations can be utilized.

Positional (Proximity) Index

• Posting List nodes may maintain position of terms in each document for Proximity search.

Apple, 3 → (D1,2, {1,5}) (D2,1, {10}) (D3,3, {1,7, 15}) …

• An alternative to phrasing
• Expands the PL storage requirements
• Using both phrase and proximity can be combined.
Term List
(Lexicon/Dictionary/Vocabulary)

• Usually we have enough memory to store the term list in memory.

• Various options
  – Sorted List: good for prefix lookup
    • Fixed length array -- wasteful
    • String of characters (primary array of integers pointing to string of terms)
    • Search tree (binary, b+trees, trie,....)
  – Hash table – with collision list; good for indexing (insert & lookup)
  – Hybrid Approach

• Can use dictionary interleaving if term index is too large (subset of terms in memory pointing to term index <term, posting> on disk )

Posting List

• Mainly resides on disk
• Brought into memory for processing
• Contiguous posting entries for each term on disk
• In memory posting:
  – Array (variable length)
  – Linked List (single link)
Memory Requirements
(single link list example)

- While in memory the posting list is not compressed.
- Typical entry

<table>
<thead>
<tr>
<th>DocID</th>
<th>tf</th>
<th>nextPointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4 bytes)</td>
<td>(2 bytes)</td>
<td>(4 bytes)</td>
</tr>
</tbody>
</table>

- For an 800,000,000 word collection, 400,000,000 posting list entries were needed (many terms did not result in a posting list entry because of stop words removal and duplicate occurrences of a term within a document).
- With 400,000,000 posting list entries, at 10 bytes per entry, we obtain a memory requirement of 4GB.

Index Construction Algorithms

All depends on the hardware availability

- Memory-based
  - Assumption: enough memory is available to construct and maintain the entire inverted index.
  - Good if enough memory and small collection
- Disk-based
  - No memory assumption; scaling to large collections
  - Various implementations exist
Memory-based Index Construction

- For each document \( d \) in the collection
  - For each term \( t \) in document \( d \)
    - Find term \( t \) in the lexicon
    - If term \( t \) exists, add a node to its posting list
    - Otherwise,
      - Add term \( t \) to the lexicon
      - Add a node to the posting list
  - After all documents have been processed, write the inverted index to disk.

Memory-based Inverted Index

- Phase I (parse and read)
  - For each document
    - Identify distinct terms in the document
    - Update, in memory the posting list for each term
- Phase II (write)
  - For each distinct term in the index
    - Write the inverted index to disk (feel free to compress the posting list while writing it)
Memory Management

- We usually don’t have more memory than the size of the document collection.
- Periodically must write inverted index to disk.
- Algorithm must be changed to periodically write to disk a subset of the inverted index $I$ and then merge the subsets.

Disk based Index Construction

(Sort/Merge-based)

- Read fixed chunk of data into memory
- Tokenize
- If needed create the term to term id mappings
- build $\langle$term, doc$\rangle$ pairs; or $\langle$term, doc, tf$\rangle$ triples; or $\langle$term and its postings$\rangle$ per implementation decisions
- Create intermediate sorted files and write on disk
- Perform m-way merging of intermediate files in memory and write onto the disk
- The outcome is one final inverted file on disk.
Disk based Index Construction
(Sort/Merge-based)

- Phase I
  - Create temp files of triples (termID, docID, tf)
- Phase II
  - Sort the triples using external mergesort
- Phase III
  - Merge the sorted triples files (2-way; m-way)
- Phase IV
  - Build Inverted index from sorted triples

Disk based Index Construction
(Sort/Merge-based)

- Phase I (parse and build temp file)
  - For each document
    - Parse text into terms, assign a term to a termID (use an internal index for this)
    - For each distinct term in the document
      - Write an entry to a temporary file with only triples <termID, docID, tf>
- Phase II (make sorted runs, to prepare for merge)
  - Do Until End of Temporary File
    - Sort the triples in memory by term id and doc id.
    - Write them out in a sorted run on disk.
Disk based Index Construction
(Sort/Merge-based)

### Phase III (merge the runs)
Repeat until there is only one run
- Merge pair-wise (2-way) or m-way sorted runs into a single run.

### Phase IV
- For each distinct term in final sorted run
  - Start a new inverted file entry.
  - Read all triples for a given term (these will be in sorted order)
  - Build the posting list (feel free to use compression)
  - Write (append) this entry to the inverted index into a binary file.
Disk based Index Construction
(Sort/Merge-based)

Sorted Run1:

<table>
<thead>
<tr>
<th>tid</th>
<th>did</th>
<th>tf</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>d1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>d2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>d1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>d2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>d1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>d1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>d1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>d2</td>
<td>3</td>
</tr>
</tbody>
</table>

Sorted Run2:

<table>
<thead>
<tr>
<th>tid</th>
<th>did</th>
<th>tf</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>d3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>d4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>d3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>d4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>d4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>d3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>d4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>d4</td>
<td>2</td>
</tr>
</tbody>
</table>

Merged:

<table>
<thead>
<tr>
<th>tid</th>
<th>did</th>
<th>tf</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>d1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>d2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>d3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>d4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>d1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>d2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>d3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>d4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>d1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>d4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>d1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>d3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>d4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>d1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>d2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>d4</td>
<td>2</td>
</tr>
</tbody>
</table>

Disk based Index Construction
(Sort/Merge-based)

Final Sorted Run:

<table>
<thead>
<tr>
<th>tid</th>
<th>did</th>
<th>tf</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>d1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>d2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>d3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>d4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>d1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>d2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>d3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>d4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>d1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>d4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>d1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>d3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>d4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>d1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>d2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>d4</td>
<td>2</td>
</tr>
</tbody>
</table>

Inverted Index

Stream of Posting List Nodes

- t1 -> d1,2, d2,1, d3,2, d4,2
- t2 -> d1,4, d2,3, d3,1, d4,2
- t3 -> d1,1, d4,1
- t4 -> d1,1, d3,3, d4,1
- t5 -> d1,2, d2,3, d4,3
Alternatives

• Instead of triples:
  – \(<term, doc>\) pairs: after sorting then create the posting with \(tf\)
  – For each term create the posting directly in memory posting
    \(<term and its postings>\) triples -- Good for dynamic collection

• Instead of term id:
  – No need for \(term id\) at all. Lexicon keeps the \(terms\)
  – No need for extra structure for the term to term id mapping

Disk-based Inverted Index Summary

• Pro
  – Not as fast as memory based, but it is scalable!

• Con
  – Requires significant additional space.
Distributed Index

- **Single index** – traditional approach
  - Use single fast machine
  - Good for some applications (enterprise search)
- **Distributed index**
  - Use several/many fast machines (servers)
  - Good for indexing tens of billions of pages (large scale)

Query Servers

- Each server has its own disk holding a portion of index
- Queries are distributed, via a centralized control, to servers that contain the related posting lists
- Common terms may map to many servers
- No single point of resource contention (*efficient*)
- If a server crashes, that portion of index is not available
Distributed Index (Cont’d)

• Web search tools access data distributed on servers worldwide but indexed centrally.

• Most of these systems have a partitioned index with a centralized control.

• Partitioning of index across multiple machines, based on terms or documents

• Using content-index, sending requests to those server that have the data

Partitioned Indexing

• Partitioning of index across multiple machines, based on either:
  • Terms (Global index organization)
    • Each node holds posting list for some terms
    • Using content-index, query terms sent to nodes having the terms
    • Higher concurrency level, but larger postings lists
  • Documents (Local index organization)
    • Each node holds a complete term index (shorter PLs)
    • Query terms sent to all nodes
    • Top k results from each node merged
    • Global statistics (e.g., idf) must be calculated

• Tiered Indexing may be used
Index Tiering

• A popular *early termination* technique to improve the efficiency of query processing

• Dividing nodes into two tiers to allocate the index of most popular documents on tier 1 and the rest on tier 2.

• Search tier 1 first, if not enough results then search tier 2.

• Note: other popular early termination techniques (*top-doc* and *query pruning*) will be discussed!

Distributed Index Construction

• Not possible on a single machine

• Various architecture for distributed indexing

• *MapReduce* architecture (a term-partitioned index)
  • Master node assigns tasks to worker nodes (*map workers* & *reduce workers*) to split up the computing jobs:
    • *Map Phase*: Parsing & building localized <term, doc> pairs
    • *Reduce Phase*: Combining/merging posting pairs for each term
MapReduce (Cont’d)

- Map & reduce phases can be done in parallel on many machines
- A map machine can be a reducer machine in the process
- Data broken into pieces (shards)...generally 16M-64 M [128M] and send to map workers as they finish their job
- Map workers work on one shard at a time (generally), unless having more than one CPU, parse and generate \(<\text{term}, \text{doc}\>\) pair (can be combined to \(<\text{term}, \text{doc}, \text{tf}\>\)
- Sort based on term, and then secondary key (doc_id)
- The same keys (terms) are assigned to the same reduce worker
- Load should be balanced on the reducers

Index in Dynamic Environment

- Data collection is not static
  - Reconstruct the index periodically from scratch (many search engines use this)
- Maintain an auxiliary index to store new document
- Maintain multiple indexes - complicated in maintaining collection statistics

Signature Files
Signature Files

- A signature is an encoding of a document, using few bits.
- Each signature may represent multiple docs.
- Thus, Two-Phase query processing:
  - Phase 1: scan signatures and identify candidate signatures
  - Phase 2: scan original text of the candidate signatures

Construction of Signatures

- Often using one or more hashing functions for each term to set a bit in a signature:
  - h(information): 0101;
  - h(retrieval): 1010;
  - h(security): 0011
- OR the term signatures of a document to build document signature
  - D1: Information retrieval: 1111
  - D2: security information: 0111
Processing of Signatures

• Boolean AND between query and document
  Q> information: 0101
    – D1: Information retrieval: 1111
    – D2: security information: 0111
  ⇒ match: D1 and D2
  Q> security: 0011
    – D1: Information retrieval: 1111
    – D2: security information: 0111
  => match: D1 and D2  - false positive (false drop)

Processing of Signatures

• Boolean AND queries: all query terms must return true
• Boolean OR queries: some query terms must return true
Signature Files Summary

- **Pros:**
  - Useful if can fit into memory
  - Easy to add or remove documents (signatures) as compared to inverted index.
  - The order of signature in the signature file does not matter.

- **Cons:**
  - Two phased processing for false matches
  - Does not rank the retrieved documents

Relational Approach will be discussed in a separate set of slides!
References