Advanced Topics in Information Retrieval

3. Efficiency & Scalability

Vinay Setty
(vsetty@mpi-inf.mpg.de)

Jannik Strötgen
(jtroetge@mpi-inf.mpg.de)
3.1. Motivation
3.2. Index Construction & Maintenance
3.3. Static Index Pruning
3.4. Document Reordering
3.5. Query Processing
3.1. Motivation
3.2. Index Construction & Maintenance
3.3. Static Index Pruning
3.4. Document Reordering
3.5. Query Processing
3.1. Motivation

- **Efficiency** is about “doing things right”, i.e., accomplishing a task using minimal resources (e.g., CPU, memory, disk).

- **Scalability** is about to be able to
  - accomplish a larger instance of a task e.g. indexing millions/billions of documents, large number of queries
  - using additional resources (e.g., faster/more CPUs, more memory/disk)
Our focus will be on two major aspects of every IR system:

- **Indexing:** how can we efficiently construct & maintain an inverted index that consumes little space.

- **Query processing:** how can we efficiently identify the top-k results for a given query without having to read posting lists completely.

Other aspects which we will not cover include:

- **Caching** (e.g., posting lists, query results, snippets)

- **Modern hardware** (e.g., GPU query processing, SIMD compression)
Hardware & Software Trends

- CPU speed has increased more than that of disk and memory: faster to read & decompress than to read uncompressed

- More memory is available; disks have become larger but not faster: now common to keep indexes in (distributed) memory

- Many (less powerful) instead of few (powerful) machines; platforms for distributed data processing (e.g., MapReduce, Spark)

- More CPU cores instead of faster CPUs; SSDs (fast reads, slow writes, wear out) in addition to HDDs; GPUs and FPGAs
Outline

3.1. Motivation
3.2. Index Construction & Maintenance
3.3. Static Index Pruning
3.4. Document Reordering
3.5. Query Processing
3.2. Index Construction & Maintenance

- **Inverted index** as widely used index structure in IR consists of
  - **dictionary** mapping terms to term identifiers and statistics (e.g., idf)
  - **posting lists** for every term recording details about its occurrences

- How to construct an inverted index from a document collection?
- How to maintain an inverted index as documents are inserted, modified, or deleted?
Observation: Constructing an inverted index (aka. inversion) can be seen as sorting a large number of (term, did, tf) tuples

- seen in (did)-order when processing documents
- needed in (term, did)-order for the inverted index

Typically, the set of all (term, did, tf) tuples does not fit into the main memory of a single machine, so that we need to sort using external memory (e.g., hard-disk drives)
Lester al. [5] describe the following algorithm by Heinz and Zobel to construct an inverted index on a single machine:

- Let $B$ be the number of (term, did, tf) tuples that fit into main memory.
- While not all documents have been processed:
  - Read (up to) $B$ tuples from the input (documents).
  - Construct in-memory inverted index by grouping & sorting the tuples.
  - Write in-memory inverted index as sorted run of (term, did, tf) tuples to disk.
- Merge on-disk runs to obtain global inverted index.
Index Construction in MapReduce

- MapReduce as a platform for distributed data processing
  - was developed at Google
  - operates on large clusters of commodity hardware
  - handles hard- and software failures transparently
  - open-source implementations (e.g., Apache Hadoop) available
  - programming model operates on key-value (kv) pairs
    - `map()` reads input data \((k_1, v_1)\) and emits kv pairs \((k_2, v_2)\)
    - platform groups and sorts kv pairs \((k_2, v_2)\) automatically
    - `reduce()` sees kv pairs \((k_2, \text{list}<v_2>)\) and emits kv pairs \((k_3, v_3)\)
Map/Reduce Example

Mappers

<table>
<thead>
<tr>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b a c a</td>
</tr>
<tr>
<td>a a c a b</td>
</tr>
<tr>
<td>b b b a a</td>
</tr>
<tr>
<td>c b a a a</td>
</tr>
<tr>
<td>a a a a a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>d2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b a d a</td>
</tr>
<tr>
<td>a a d a b</td>
</tr>
<tr>
<td>b b b a a</td>
</tr>
<tr>
<td>d b a a a</td>
</tr>
</tbody>
</table>

Intermediate Sorting/combining

{a, <d1,16>}
{b, <d1, 6>}
{c, <d1, 3>}

{a, <d1,16>, <d2,11>}
{b, <d1, 6>, <d2, 6>}
{c, <d1, 3>}
{d, <d2, 3>}

Reducers
Index Construction in MapReduce

\textbf{map}(\text{did, list<term>})

\begin{verbatim}
map<term, integer> tfs = new map<term, integer>();
// determine term frequencies
for each term in list<term>:
    tfs.adjustCount(term, +1);
// emit postings
for each term in tfs.keys():
    emit (term, (did, tfs.get(term)));
\end{verbatim}

// platform groups & sorts output of map phase by term

\textbf{reduce}(\text{term, list<(did, tf)>})

\begin{verbatim}
// emit posting list
emit (term, list<(did, tf)>)
\end{verbatim}
Index Maintenance

- Document collections are **not static**, but documents are **inserted, modified, or deleted** as time passes; changes to the document collection should quickly be visible in search results

- **Typical approach**: Collect changes in main memory
  - deletion list of deleted documents
  - in-memory delta inverted index of inserted and modified documents
  - process queries over both the on-disk global and in-memory delta inverted index and filter out result documents from the deletion list

- What if the available main memory has been exhausted?
Rebuild

- Rebuild the on-disk global index from scratch
  - in a separate location; switch over to new index once completed
- attractive for small document collections
- attractive when document deletions are common
- requires re-processing of entire document collection
- easy to implement
Merge

- **Merge** the on-disk global index with the in-memory delta index
  - in a *separate location*; switch over to new index once completed
  - for each term, **read** posting lists from on-disk global index and in-memory delta index, **merge** them, **filter out** deleted documents, and **write** the merged posting list to disk
  - requires **reading entire on-disk global index**

- **Analysis:** Let $B$ be capacity of the in-memory delta index (in terms of postings) and $N$ be the total number of postings
  - $N / B$ merge operations each having cost $O(N)$
  - total cost is in $O(N^2)$
Lester et al. [5] propose to partition the inverted index into index partitions of geometrically increasing sizes:

- tunable by parameter \( r \)
- index partition \( P_0 \) is in **main memory** and contains up to \( B \) postings
- index partitions \( P_1, P_2, \ldots \) are **on disk** with capacity invariants
  - partition \( P_j \) contains at most \( (r-1) r^{j-1} B \) postings
  - partition \( P_j \) is either empty or contains at least \( r^{j-1} B \) postings
- whenever \( P_0 \) overflows, a **merge** is triggered

Query processing has to access all (non-empty) partitions \( P_i \), leading to higher cost due to required disk seeks.
Fig. 3. The merging pattern established when $r = 3$. The first index is placed into partition 3 only after nine bufferloads have been generated by the in-memory part of the indexing process. All numbers listed represent multiples of $b$, the size of each bufferload.

and, as before, the stream of arriving documents is processed in fixed buffer-loads of $b$ documents. The first bufferload of pointers is placed, without change, into partition 1. The second bufferload of pointers can be merged with the first, still in partition 1, to make a partition of $2b$ pointers. But the third bufferload of pointers cannot be merged into partition 1, because doing so would violate the $(r - 1)b = 2b$ limit on partition 1. Instead, the 3 pointers that are the result of this merge are placed in partition 2, and partition 1 is cleared. The fourth bufferload of pointers must be placed in partition 1, because it cannot be merged into partition 2. The fifth joins it, and then the sixth bufferload triggers a three-way merge, to make a partition containing $6b$ pointers in the second partition. Figure 3 continues this example, and shows how the concatenation of three more bufferloads of pointers from the in-memory part of the index leads to a single index of $9b$ pointers in the third partition.

5.2 Analysis

Within each partition the index sizes follow a cyclic pattern that is determined by the radix $r$. For example, in Figure 3, the “Partition 2” column cycles through sizes 0, 3, 6, and then repeats. In general, the $j$th partition of an index built with radix $r$ cycles through the sequence $0, rj - 1, 2rj - 1, \ldots, (r - 1)rq - 1$.

Geometric Merge

- **Analysis**: Let $B$ be the capacity of the in-memory partition $P_0$ and $N$ be the total number of postings
  - there are at most $1 + \lceil \log_r(N/B) \rceil$ partitions
  - each posting merged at most once into each partition
  - total cost is $O(N \log N/B)$
Logarithmic Merge

- **Logarithmic merge** is a simplified variant of geometric merge
  - partition $P_0$ is in **main memory** and contains $B$ postings
  - partition $P_1$ is on disk and contains up to $2B$ postings
  - partition $P_2$ is on disk and contains up to $4B$ postings
  - partition $P_j$ is on disk and contains up to $2^j B$ postings
  - whenever $P_0$ overflows, a cascade of merges is triggered

- Log-structured merge tree (LSM-Tree) prominent in database systems (e.g., to manage logs) is based on the same principle
Wu et al. [9] use the log-structured inverted index to support high update rates when indexing social media.
Index Management in Elasticsearch

- Indexes are stored as shards
  - Each index has a fixed number of shards
  - By default 5 shards per index - primary shards
- Shards are replicated
  - Each primary shard is replicated
  - Replication factor is a parameter
- Why shards?
  - Load balance
  - Distribution
  - Fault tolerance

A shard is a fully contained horizontal partition of index
Index Management in Elasticsearch

Node 1

Node 2

Node 3

Elasticsearch cluster

R1

P0

R0

R1

P1

R0

D1

D2
Elasticsearch Shards

- Shards are immutable
- Insert only!
- New documents are added to smaller segments
- When segments grow they are merged
Elasticsearch Shards

Segment

Dictionary

a ..... g ..... z

d_{123}, 2  d_{125}, 2  d_{227}, 1

Posting list
Lucene Dynamic Indexing

- Segments in Lucene are immutable
  - Cannot be changed
  - Can be created, merged and deleted
- When new documents are added
  - Small segments are created
  - When number of segments grow
  - Some merging technique is used such as logarithmic merging
Dynamic Indexing

Main Memory Buffer

D1  D2  D3  D4  D5  D6  D7
## Lucene Segment Merging (Insert only)

<table>
<thead>
<tr>
<th>Size</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 GB</td>
<td>0 sec</td>
</tr>
<tr>
<td>500 MB</td>
<td>4.1 MB</td>
</tr>
<tr>
<td></td>
<td>1 segs; _0</td>
</tr>
<tr>
<td></td>
<td>0.0 MB merging</td>
</tr>
<tr>
<td></td>
<td>0.0 MB merged</td>
</tr>
<tr>
<td>100 MB</td>
<td></td>
</tr>
<tr>
<td>50 MB</td>
<td></td>
</tr>
<tr>
<td>10 MB</td>
<td></td>
</tr>
</tbody>
</table>
Lucene Dynamic Indexing

- **How do deletes work?**
- **When documents are deleted**
  - They are marked deleted in the segments
- **When are they purged?**
Lucene Segment Merging with Deletions

Query Processing in Elasticsearch

Node 1

Node 2

Node 3

Query
Outline

3.1. Motivation
3.2. Index Construction & Maintenance
3.3. Static Index Pruning
3.4. Document Reordering
3.5. Query Processing
3.3. Static Index Pruning

- Static index pruning is a form of **lossy compression** that
  - removes postings from the inverted index
  - allows for **control of index size** to make it fit, for instance, into main memory or on low-capacity device (e.g., smartphone)

- **Dynamic index pruning**, in contrast, refers to query processing methods (e.g., WAND or NRA) that avoid reading the entire index
Term-Centric Index Pruning

- Carmel et al. [3] propose **term-centric** static index pruning

**Idea**: Remove postings from posting list for term $v$ that are unlikely to contribute to top-$k$ result of query including $v$

**Algorithm**: For each term $v$

- determine $k$-th highest score $z_v$ of any posting in posting list for $v$
- remove all postings having a score less than $\varepsilon \cdot z_v$

Despite its simplicity the method guarantees for any query $q$ consisting of $|q| < 1 / \varepsilon$ terms a “close enough” top-$k$ result
Document-Centric Index Pruning

- Büttcher and Clarke [2] propose document-centric index pruning

- Idea: Remove postings for document $d$ corresponding to non-important terms for which it is unlikely to be in the query result

- Importance of term $v$ for document $d$ is measured using its contribution to the KL divergence from background model $D$

\[ P[v \mid \theta_d] \log \left( \frac{P[v \mid \theta_d]}{P[v \mid \theta_D]} \right) \]

- $\text{DCP}_{\text{Const}}$ selects constant number $k$ of postings per document

- $\text{DCP}_{\text{Rel}}$ selects a percentage $\lambda$ of postings per document
Term-Centric vs. Document-Centric

- Büttcher and Clarke [3] compare term-centric (TCP) and document-centric (DCP) index pruning on TREC Terabyte
  - Okapi BM25 as baseline retrieval model
  - on-disk inverted index: 12.9 GBytes, 190 ms response time
  - pruned in-memory inverted index: 1 GByte, 18 ms response time

<table>
<thead>
<tr>
<th>[ TREC 2004 Terabyte queries (topics 701-750) ]</th>
<th>BM25 Baseline</th>
<th>DCP(^{(\lambda=0.062)}_{\text{Rel}})</th>
<th>DCP(^{(k=21)}_{\text{Const}})</th>
<th>TCP(^{(k=24500)}_{(n=16000)})</th>
</tr>
</thead>
<tbody>
<tr>
<td>P@5</td>
<td>0.5224</td>
<td>0.5020</td>
<td>0.4735</td>
<td>0.4490*</td>
</tr>
<tr>
<td>P@10</td>
<td>0.5347</td>
<td>0.4837</td>
<td>0.4755</td>
<td>0.4347*</td>
</tr>
<tr>
<td>P@20</td>
<td>0.4959</td>
<td>0.4490</td>
<td>0.4224</td>
<td>0.4163</td>
</tr>
<tr>
<td>MAP</td>
<td>0.2575</td>
<td>0.1963</td>
<td>0.1621**</td>
<td>0.1808</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[ TREC 2005 Terabyte queries (topics 751-800) ]</th>
<th>BM25 Baseline</th>
<th>DCP(^{(\lambda=0.062)}_{\text{Rel}})</th>
<th>DCP(^{(k=21)}_{\text{Const}})</th>
<th>TCP(^{(k=24500)}_{(n=16000)})</th>
</tr>
</thead>
<tbody>
<tr>
<td>P@5</td>
<td>0.6840</td>
<td>0.6760</td>
<td>0.6000**</td>
<td>0.5640**</td>
</tr>
<tr>
<td>P@10</td>
<td>0.6400</td>
<td>0.5980</td>
<td>0.5300*</td>
<td>0.5380**</td>
</tr>
<tr>
<td>P@20</td>
<td>0.5660</td>
<td>0.5310</td>
<td>0.4560**</td>
<td>0.4630**</td>
</tr>
<tr>
<td>MAP</td>
<td>0.3346</td>
<td>0.2465</td>
<td>0.1923**</td>
<td>0.2364</td>
</tr>
</tbody>
</table>
Outline

3.1. Motivation
3.2. Index Construction & Maintenance
3.3. Static Index Pruning
3.4. Document Reordering
3.5. Query Processing
Index Compression

- Sequences of non-decreasing integers (here: document identifiers) in posting lists are compressed using
  - delta encoding representing elements as difference to predecessor
  
  \[ \langle 1, 7, 11, 21, 42, 66 \rangle \rightarrow \langle 1, 6, 4, 10, 21, 24 \rangle \]

- Variable-byte encoding: (aka. 7-bit encoding) represents integers (e.g., deltas of term offsets) as sequences of 1 continuation + 7 data bits

<table>
<thead>
<tr>
<th>docIDs</th>
<th>gaps</th>
<th>VB Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>624</td>
<td>0</td>
<td>00000100 1110000</td>
</tr>
<tr>
<td>629</td>
<td>5</td>
<td>10000101</td>
</tr>
<tr>
<td>914</td>
<td>285</td>
<td>00000100 10011101</td>
</tr>
</tbody>
</table>

- Gamma encoding: unary code to represent length followed by offset binary of an integer but with leading 1 removed
  - e.g. 13 = 1101 = 1110101
3.4 Document Reordering

- Document reordering methods seek to improve compression effectiveness by assigning document identifiers so as to obtain small gaps.

- Content based document reordering

- K-means clustering
  - similar documents get closer document ids

- K-Scan
  - Single scan k-means

- URL-based document id assignment
Content-Based Document Reordering

- Silvestri et al. [7] develop methods for the scenario when only document contents are available but no meta-data (e.g., URL)

- **Intuition**: Similar documents, having many terms in common, should be assigned numerically close document identifiers

- **Documents** are modeled as **sets** (not bags) of terms

- **Document similarity** is measured using the **Jaccard coefficient**

\[
J(d_i, d_j) = \frac{|d_i \cap d_j|}{|d_i \cup d_j|}
\]
Algorithm: TDAssign(document collection $D$)

// split $D$ into equal-sized partitions $D_L$ and $D_R$

pick representatives $d_L$ and $d_R$ (e.g., randomly)

if $(|D_L| \geq |D| / 2) \lor (|D_R| \geq |D| / 2)$

assign $d$ to smaller partition

else if $J(d, d_L) > J(d, d_R)$

assign $d$ to $D_L$

else

assign $d$ to $D_R$

return $\text{TDAssign}(D_L) \oplus \text{TDAssign}(D_R)$

TDAssign has time complexity in $O(|D| \log |D|)$
\textbf{kScan}

- **Algorithm**: \texttt{kScan(document collection D)}
  
  // split D into k equal-sized partitions D\textsubscript{i}

  \begin{algorithm}
  \texttt{n = |D|}
  \texttt{for i = 1 \ldots k}
  \hspace{1em} \texttt{d\textsubscript{i} = longest document from D}
  \hspace{1em} \texttt{assign n/k documents with highest similarity J(d, d\textsubscript{i}) to D\textsubscript{i}}
  \hspace{1em} \texttt{D = D \setminus D\textsubscript{i}}
  \texttt{return < d from D\textsubscript{1}> \oplus \ldots \oplus <d from D\textsubscript{k}>}
  \end{algorithm}

- \texttt{kScan} has \textit{time complexity} in $O(k \cdot |D|)$

- \texttt{kScan} \textbf{outperforms} \texttt{TDAssign} in terms of compression effectiveness (bits per posting) in experiments on collections of web documents
URL-Based Document Reordering


- **Intuition:** Documents with lexicographically close URLs tend to have similar contents (e.g., www.x.com/a and www.x.com/b).

- **Algorithm:**
  - sort documents lexicographically according to their URL
  - assign consecutive document identifiers (1 \(\ldots\) |D|)
Silvestri [8] reports experiments conducted on a large-scale crawl of the Brazilian Web (about 6 million documents)

<table>
<thead>
<tr>
<th></th>
<th>VByte</th>
<th>Gamma</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>11.40</td>
<td>12.72</td>
<td>12.71</td>
</tr>
<tr>
<td>URL</td>
<td>9.72</td>
<td>7.72</td>
<td>7.69</td>
</tr>
<tr>
<td>kScan</td>
<td>9.81</td>
<td>8.82</td>
<td>8.80</td>
</tr>
</tbody>
</table>

URL-based document ordering outperforms content-based document ordering (kScan), requiring fewer bits per posting on average
3.1. Motivation
3.2. Index Construction & Maintenance
3.3. Static Index Pruning
3.4. Document Reordering
3.5. Query Processing
Query Processing

- Query processing methods operate on inverted index
  - holistic query processing methods determine the full query results (e.g., document-at-a-time and term-at-a-time)
  - top-k query processing methods (aka. dynamic index pruning) determine only the top-k query result and avoid reading posting lists completely
    - Fagin’s TA and NRA for score-ordered posting lists
    - WAND and Block-Max WAND for document-ordered posting lists
Broder et al. [1] describe WAND (weak AND) as a top-k query processing method for document-ordered posting lists.

- **DAAT-style traversal of posting lists in parallel**
- Assumes that the maximum score \( \max(i) \) per posting list is known.
- **Pivoted cursor movement** based on current top-k result.
  - Let \( \min_k \) denote the worst score in the current top-k result (1).
  - Sort cursors for posting lists based on their current document identifier \( \text{cdid}(i) \) (2).
  - Pivot document identifier \( p \) is the smallest \( \text{cdid}(j) \) such that (3)

  \[
  \min_k < \sum_{i \leq j} \max(i)
  \]

  - Move all cursors \( i \) with \( \text{cdid}(i) < p \) up to pivot \( p \)
Example: Pivoted cursor movement based on top-1 result

- **Top-1**
  - \(d_1: 8\)
  - \(a\) \(d_1, 2\) \(\ldots\) \(d_3, 1\) \(\ldots\) \(\max(a) = 3\)
  - \(b\) \(d_1, 3\) \(\ldots\) \(d_2, 3\) \(\ldots\) \(\max(b) = 3\)
  - \(c\) \(d_1, 3\) \(\ldots\) \(d_9, 3\) \(\ldots\) \(\max(c) = 3\)

- It is safe to move the cursor for posting lists \(a\) and \(b\) forward to \(d_9\)

\[
\begin{align*}
\text{cdid} & \quad \Sigma \\
\hline
d_2, 3 & 3 \\
d_3, 1 & 6 \\
d_9, 3 & 9 \\
\end{align*}
\]

- \(\min_k = 8\)
- \(p = d_9\)

(1) (2) (3)
Ding and Suel [4] propose the block-max inverted index:

- Store posting list as a sequence of compressed posting blocks.
- Each block contains a fixed number of postings (e.g., 64).
- Keep the minimum document identifier and maximum score per block.

\[
\begin{array}{ccc}
(1, 5) & (7, 2) & (11, 3) \\
\text{a} & d_1, 2 & d_3, 5 & d_7, 2 & d_9, 1 & d_{11}, 3 & d_{13}, 2 \\
\end{array}
\]

\[\max(a) = 5\]

These are available without having to decompress the block.
Block-Max WAND

- Pivoted cursor movement considering per-block maximum scores
  - determine **pivot** $p$ according to WAND
  - perform shallow cursor movement for all cursors $i$ with $\text{cdid}(i) < p$
    (i.e., do not decompress if a new posting block is reached)
  - if any document from current blocks can make it into top-$k$, i.e.:
    \[
    \min_k < \sum_{i: \text{cdid}(i) \leq p} \text{block}_\text{max}(i)
    \]
    perform deep cursor movement (i.e., decompress posting blocks) and continue as in WAND
  - **else** move cursor with minimal $\text{cdid}(i)$ to
    \[
    \min \left( \min_{i: \text{cdid}(i) \leq p} \text{next}\_\text{block}\_\text{mdid}(i), \text{cdid}(p + 1) \right)
    \]
Example: Pivoted cursor movement based on top-1 result

Top-1
\[ d_1 : 8 \]

**a**  d\(_1\), 2  ...  d\(_3\), 1  ...  (2, 1)  (5, 1)  (11, 3)  \[ \text{shallow} \]

**b**  d\(_1\), 3  ...  d\(_2\), 3  ...  (2, 3)  (4, 1)  (10, 2)  \[ \text{shallow} \]

**c**  d\(_1\), 3  ...  d\(_9\), 3  ...  (7, 3)  (14, 1)  (17, 2)

**d**  d\(_2\), 3  ...  d\(_{11}\), 3  ...  \[ \text{shallow} \]
Summary

- **Inverted indexes** can be efficiently constructed offline by using external memory sort or MapReduce.
- **Inverted indexes** can be efficiently maintained by using logarithmic/geometric partitioning.
- Index maintenance and query processing in Elasticsearch.
- **Static index pruning methods** reduce index size by systematically removing postings.
- **Document reordering methods** reduce index size by assigning document identifiers so as to yield smaller gaps.
- **Query processing** on document-ordered inverted indexes can be greatly sped up by pivoted cursor movement as part of WAND and Block-Max WAND.
References


Additionally you can also refer to Chapter 5 in Introduction to Information retrieval by Christopher D. Manning et.al.
Some slides were borrowed from Prof. Klaus Berberich