Part XI

Search
Ever-increasing amounts of data are available electronically. These data have varying degrees of **structure**.

- (R)DBMS
- Social graphs
- Web pages
- Unstructured text
- XML
- Text with markup

How can we efficiently store and access such **un-structured data**?

→ success of **search engines** "search"
Let’s start with what we have...

- *E.g.*, four **documents**

<table>
<thead>
<tr>
<th>doc₁</th>
<th>doc₂</th>
<th>doc₃</th>
<th>doc₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.</td>
<td>Fishkeepers often use the term tropical fish to refer only those requiring freshwater, with salt-water tropical fish referred to as marine fish.</td>
<td>Tropical fish are popular aquarium fish, due to their often bright coloration.</td>
<td>In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.</td>
</tr>
</tbody>
</table>

- Say we’re interested in “freshwater fish.”
  → Two **search terms**: “freshwater” and “fish”
Boolean Queries

Query in SQL-style notation:

```sql
SELECT *
FROM Documents AS D
WHERE D.content CONTAINS 'freshwater'
AND D.content CONTAINS 'fish'
```

Idea:
- **Index** to look up *term* → *document*.
  - There will be an index entry for every word in every document.

💡 Execution strategy for the above query?
Boolean Queries

Discussion:

- Returns all documents that contain both search terms.
  - This may be more than we want.

  Google: about 21 million pages with “freshwater” and “fish!”

- Returns nothing else.
  - This may be less than we want.

  \[doc_2\] and \[doc_3\] may be relevant for us, too.

- Returns documents in no specific order.
  - But some documents might be more relevant than others.

  \[ORDER BY\] won’t help!

Boolean Query: (exact match retrieval)

- A predicate precisely tells whether a document belongs to the result.

Ranked Query:

- Results are ranked according to their relevance (to the query).
Goal: Rank documents higher that are closer to the query’s intention.

→ Extract features from each document.
→ Use feature vector and query to compute a score.
Idea:
- Compute similarity between query and document.

Similarity:
- Define a set of features to use for ranking.
  - each term in the collection is one feature
  - possible features: document size/age, page rank, etc.
- For each document compute a feature vector $d_i$;
  - e.g., yes/no features; term count; etc.
- For the query compute a feature vector $q$.
- Measure similarity of the two vectors.
Two vectors are similar if the **angle** between them is small.

![Vector Space Model Diagram](image)

**Cosine** between $d_i$ and $q$:

$$\cos(d_i, q) = \frac{\sum_j d_{ij} \cdot q_j}{\sqrt{\sum_j d_{ij}^2 \cdot \sum_j q_j^2}}$$

($j$ iterates over all features/terms; $i$ is the document in question)

→ “vector space model”
Ignoring the normalization term: \( \text{sim}(d_i, q) = \sum_j d_{ij} q_j \).

→ Multiply corresponding feature values, then sum up.

Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

\[
\begin{array}{l}
\text{document} \\
9.7 \text{ fish} \quad \leftrightarrow \quad \text{fish} \ 5.2 \\
4.2 \text{ tropical} \quad \leftrightarrow \quad \text{tropical} \ 3.4 \\
22.1 \text{ tropical fish} \quad \leftrightarrow \quad \text{tropical fish} \ 9.9 \\
8.2 \text{ freshwater} \quad \leftrightarrow \quad \text{chichlids} \ 1.2 \\
2.3 \text{ species} \quad \leftrightarrow \quad \text{barbs} \ 0.7 \\
\text{topical features} \quad \leftrightarrow \quad \text{topical features} \\
14 \text{ incoming links} \quad \leftrightarrow \quad \text{incoming links} \ 1.2 \\
3 \text{ days last upd.} \quad \leftrightarrow \quad \text{days last upd.} \ 0.9 \\
\text{quality features} \quad \leftrightarrow \quad \text{quality features} \\
303.01 \text{ document score}
\end{array}
\]

What does this mean for an implementation?
What are good features (and their values)?

Topical Features:
- Each term in the collection (∼ vocabulary) is one feature.

Feature Value:
- A document with multiple occurrences of ‘foo’ is likely more relevant to queries that contain ‘foo’.
  → term frequency $tf$ as a feature value.

$$tf_{doc,foo} = \frac{\text{number of occurrences of ‘foo’ in } doc}{\text{number of words in } doc}$$

→ Normalize to account for different document sizes.
Terms that occur in many documents are less discriminating.

→ inverse document frequency \( idf \):

\[
idf_{\text{foo}} = \log \frac{\text{number of documents in the collection}}{\text{number of documents that contain ‘foo’}}
\]

→ \( idf \) is a property of the term, not the document!

Combine to obtain feature value \( d_{ij} \) (document \( i \), term \( j \)):

\[
d_{ij} = tf_{ij} \cdot idf_j
\]

Do the same thing for query features \( q_j \).
tf/idf weights essentially come from intuition and experiments.

→ No formal basis for the formulas above.

Alternative Formulations:

- **Boolean “frequencies”:**

  \[
  tf_{ij} = \begin{cases} 
  1 & \text{when term } j \text{ occurs in document } i \\
  0 & \text{otherwise}
  \end{cases}
  \]

- Use **logarithm** rather than raw count:

  \[
  tf_{ij} = \log(f_{ij}) + 1
  \]

  (add 1 to ensure non-zero weights)

- Give benefit for words that occur in titles, etc.
Quality Features

Some document characteristics do not tell whether the document matches the subject of a query.

→ Yet they may be relevant to the ranking/quality of the document.

Examples:

- Web pages with higher incoming link count may more trustworthy.
- Documents that weren’t modified for a long time may contain outdated information.

Quality features for the query may help to express the user’s intention:

- Is (s)he only interested in the most recent news?
  → Give higher weight to features like ‘days last updated’.
PageRank\textsuperscript{28} is a quality feature that became popular with the rise of Google.

**Motivation:** Use link analysis to rate the popularity of a web site.

→ **Incoming links** indicate quality, but are easy to manipulate.
→ Try to weigh each incoming link by the popularity of the originating site.

**Idea:**

- Assume a **random Internet surfer** Alice.
  → On every page, randomly click some of its outgoing links.
  → Every now and then (with probability $\lambda$) jump to a random page instead.
- PageRank of a page $p$: What is the probability that Alice looks at $p$ when we randomly interrupt her browsing?

\textsuperscript{28}Named after Google founder Larry Page.
Computing PageRank

Example:

Probability that Alice ends up on $C$:

$$PR(C) = \lambda \cdot \frac{1}{3} + (1 - \lambda) \cdot \left( \frac{PR(A)}{2} + \frac{PR(B)}{1} \right).$$

Generally:

$$PR(u) = \frac{\lambda}{N} + (1 - \lambda) \cdot \sum_{v \in B_u} \frac{PR(v)}{\text{outgoing}_v}.$$
But we don’t know $PR(A)$ and $PR(B)$, yet!

→ **Iterate** the above formula and PageRanks will converge.

→ *E.g.*, initialize with equal PageRanks $1/N$.

- A typical value for $\lambda$ is 0.15.
- Today, PageRank is just one out of many features used in ranking.
  → Tends to have most impact on popular queries.
Prepare for Queries

Before querying, documents must be analyzed:

1. **Parse** and **tokenize** document.
   - Strip markup (if applicable), identify text to index.
   - Break text into **tokens** (words).
   - Normalize **capitalization**.

2. **Remove stop words**.
   - ‘the,’ ‘a,’ ‘this,’ ‘that,’ etc. generally not useful for search.

3. Normalize words to terms (**stemming**).
   - *E.g.*, ‘fishing,’ ‘fished,’ ‘fisher’ → ‘fish’
   - Stems need not themselves be words (*e.g.*, ‘describe,’
     ‘describing,’ ‘description’ → ‘describ’)

4. Some systems also extract **phrases**.
   - *E.g.*, ‘european union,’ ‘database conference’

Terms are then used to populate an **index**.
Inverted Files

A search engine’s document collection is essentially a mapping

\[ \text{document} \rightarrow \text{list of } \text{term} \ . \]

To search the collection, it is much more useful to construct the mapping

\[ \text{term} \rightarrow \text{list of } \text{document} \ . \]

E.g.,

<table>
<thead>
<tr>
<th>term</th>
<th>docs</th>
<th>term</th>
<th>docs</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>(\textit{doc}_1)</td>
<td>both</td>
<td>(\textit{doc}_1)</td>
</tr>
<tr>
<td>aquarium</td>
<td>(\textit{doc}_3)</td>
<td>bright</td>
<td>(\textit{doc}_3)</td>
</tr>
<tr>
<td>are</td>
<td>(\textit{doc}_3, \textit{doc}_4)</td>
<td>coloration</td>
<td>(\textit{doc}_3, \textit{doc}_4)</td>
</tr>
<tr>
<td>around</td>
<td>(\textit{doc}_1)</td>
<td>derives</td>
<td>(\textit{doc}_4)</td>
</tr>
<tr>
<td>as</td>
<td>(\textit{doc}_2)</td>
<td>due</td>
<td>(\textit{doc}_3)</td>
</tr>
</tbody>
</table>
A representation of this type is thus also called inverted file\textsuperscript{29}.

- Conceptually, an inverted file is the same as a database index.
- However, in a search engine, the inverted file forms the heart of the whole system.
  
  → It makes sense to specialize and fine-tune its implementation.
  → Terminology: For each index term there’s one inverted list. The inverted list is a list of postings.

Characteristics:

- The set of index terms is pretty much fixed (e.g., given by the English dictionary).
- Sizes of inverted lists, by contrast, grow with the number of documents indexed.
  
  → Their sizes typically follow a Zipfian distribution.

\textsuperscript{29}sometimes also “inverted index”
Inverted files can grow **large**.

→ One posting for every term in every document.
→ Index about as large as entire document collection.

It thus makes sense to **compress** inverted lists.

_predicate: How well will lists of document ids compress?

© Jens Teubner · Architecture & Implementation of DBMS · Summer 2018
Inverted Files—Compression

This changes if we **sort**, then **delta-encode** inverted lists:

\[1, 5, 9, 18, 23, 24, 30, 44, 45, 48\]

\[\downarrow\]

\[1, 4, 4, 9, 5, 1, 6, 14, 1, 3\]

Can now use compression schemes that favor **small values**.

→ **E.g., null suppression**
  - Suppress **leading null bytes**.
  - Encode number of suppressed nulls with fixed-length prefix.
  - *E.g.,* 18 \(\rightarrow\) 000010010; 427 \(\rightarrow\) 010000000110101011.

→ **E.g., unary codes**
  - Encode \(n\) with sequence of \(n\) 1s, followed by a 0.
  - *E.g.,* 0 \(\rightarrow\) 0; 1 \(\rightarrow\) 10; 2 \(\rightarrow\) 110; 12 \(\rightarrow\) 11111111111110.
Inverted Files—Elias-γ Compression

Elias-γ Codes:

- To encode \( n \), compute

\[
\begin{align*}
n_d &= \lfloor \log_2 n \rfloor \quad \text{“position of leading bit”} \\
n_r &= n - 2^{\lfloor \log_2 n \rfloor} \quad \text{“value encoded by remaining bits”}
\end{align*}
\]

- Then, represent \( n \) using
  - \( n_d \), unary-encoded; followed by
  - \( n_r \), binary-encoded.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( n_d )</th>
<th>( n_r )</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>10 0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>10 1</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>7</td>
<td>1110 111</td>
</tr>
<tr>
<td>255</td>
<td>7</td>
<td>127</td>
<td>111111110 1111111</td>
</tr>
</tbody>
</table>
PFOR Compression:

Illustrated here using compressed representation of the digits of π.\textsuperscript{30}

\begin{align*}
\text{header} & \quad 3 & 1 \\
4 & 1 & 5 & \bot & 2 & 6 & 5 & 3 & 5 & \bot \\
\bot & \bot & \bot & 3 & 2 \\
9 & 7 & 9 & 8 & 9
\end{align*}

Decompressed numbers: 31415\textcolor{red}{9}26535897932

\textsuperscript{30}PFOR was developed in the context of the MonetDB/X100 main-memory database project, now commercialized by Actian.
PFOR Decompression

During decompression, we have to consider all the exceptions:

```c
for (i=j=0; i<n; i++)
    if (code[i] != ⊥)
        output[i] = DECODE (code[i]);
    else
        output[i] = exception[--j];
```

For PFOR, \texttt{DECODE} is a simple addition:

```c
#define DECODE(a) ((a) + base\_value)
```

Problem on modern hardware: High branch misprediction cost.
PFOR: Avoiding the Misprediction Cost

Invest some unnecessary work to avoid high misprediction penalty.

Run decompression in **two phases**:

1. **Decompress** all regular fields, but don’t care about exceptions.
2. Work in all the exceptions and **patch** the result.

```c
/* ignore exceptions during decompression */
for (i = 0; i < n; i++)
    output[i] = DECODE(code[i]);

/* patch the result */
foreach exception
    patch corresponding output item;
```
We **don’t** want to use a branch to find all exception targets!

**Thus:** interpret values in “exception holes” as **linked list**:

→ Can now traverse exception holes and patch in exception values.
The resulting decompression routine is branch-free:

```c
/* ignore exceptions during decompression */
for (i = 0; i < n; i++)
    output[i] = DECODE (code[i]);

/* patch the result (traverse linked list) */
j = 0;
for (cur = first_exception; cur < n; cur = next) {
    next = cur + code[cur] + 1;
    output[cur] = exception[--j];
}
```
With inverted lists available, the evaluation of

\[ \text{term}_1 \text{ and term}_2 \]

amounts to computing the **intersection** of the two inverted lists.

**Strategy:** (assuming inverted lists are **sorted** by document id)

- “Merge” lists \( l_{\text{term}_1} \) and \( l_{\text{term}_2} \) (↗ merge\_join (), slide 186).
- Cost: linear scan of \( l_{\text{term}_1} \) plus linear scan of \( l_{\text{term}_2} \).

**Problem:** Long, inefficient scans

*E.g.,*

- \( |l_{\text{fish}}| = 300 \text{ M}; \ |l_{\text{freshwater}}| = 1 \text{ M}. \)
- At least 299 M \( l_{\text{fish}} \) entries scanned unnecessarily.

→ **Skip** over those entries?
Skip Pointers

Idea:

- **Skip pointers** point to every $k$th posting.
- skip pointer: $\langle \text{byte pos}, \text{doc id} \rangle$.

**Skip forward to document $d$:**

1. Read skip pointer list as long as $\text{doc id} \leq d$.
2. Follow the pointer and scan posting list from there to find $d$. 
Skip Pointers

**Example:** $|l_{fish}| = 300$ M; $|l_{freshwater}| = 1$ M; skip distance $k$.

For complete merge: (cost to read $l_{fish}$)

- Read all $300$ M/$k$ skip pointers.
- Perform $1$ M posting list scans; average length: $\frac{1}{2}k$.
- Total cost to read $l_{fish}$: $300,000,000/k + 500,000k$.
Skip Pointers

**Improvements:**

- Rather than reading skip pointer list sequentially, use
  → binary search,
  → exponential search (also: “galloping search”), or
  → interpolation search.

**Why not use these search methods directly on the inverted list?**
Query Execution (with Ranking)

Idea:

1. **Compute score** for each document.
2. **Sort** by score.
3. **Return** top $n$ result documents.

Only features $j$ where $q_j \neq 0$ will contribute to $\sum_j d_{ij} q_j$.

→ Score only documents that appear in at least one inverted list for the index terms in $q$. 
Term-at-a-Time Retrieval

Process inverted lists one after another:

1. \( R \leftarrow \text{PriorityQueue}\left(n\right) \);
2. \( A \leftarrow \text{HashTable}\left()\) ;
3. \textbf{foreach} term \( j \) \textbf{in} \( q \) \textbf{do}
   4. \quad \textbf{foreach} document \( i \) \textbf{in} inverted list for \( j \) \textbf{do}
   5. \quad \quad \text{score} \leftarrow A.\text{get}\left(i\right) ;
   6. \quad \quad \textbf{if} \ not \ found \textbf{then}
   7. \quad \quad \quad A.\text{put}\left(i, d_{ij}q_j\right) ;
   8. \quad \quad \textbf{else}
   9. \quad \quad \quad A.\text{put}\left(i, \text{score} + d_{ij}q_j\right) ;
10. \textbf{foreach} \( \langle i, \text{score} \rangle \) \textbf{in} \( A \) \textbf{do}
11. \quad R.\text{add}\left(i, \text{score} \right) ;
12. \textbf{return} \( R \) ;
Document-at-a-Time Retrieval

1. $R \leftarrow \text{PriorityQueue}(n)$;
2. \textbf{foreach} term $j$ in $q$ \textbf{do}
3. \hspace{1em} $L$.add (inverted list for $j$);
4. \textbf{while} $L$ is not empty \textbf{do}
5. \hspace{2em} /* Find next document $i$ in any inverted list */
6. \hspace{3em} $i \leftarrow \text{smallest } l_j.dociD \text{ in } L$;
7. \hspace{2em} /* Score document $i$ */
8. \hspace{3em} $score \leftarrow 0$;
9. \hspace{2em} \textbf{foreach} $l_j \in L$ \textbf{do}
10. \hspace{3em} \hspace{1em} \textbf{if} $l_j.dociD = i$ \textbf{then}
11. \hspace{4em} \hspace{1em} $score \leftarrow score + d_{ij}q_j$;
12. \hspace{4em} \hspace{1em} $l_j$.advance () ;
13. \hspace{4em} \hspace{1em} \textbf{if} eof ($l_j$) \textbf{then}
14. \hspace{4em} \hspace{2em} $L$.remove ($l_j$) ;
15. \hspace{3em} $R$.add ($i$, $score$) ;
16. \hspace{2em} \textbf{return} $R$ ;
Restriction:
- Return only documents that contain all of the query terms.

Then:
- Document-at-a-time $\rightarrow$ intersection/merging.
  $\rightarrow$ Use skip lists to navigate through inverted lists quickly.
- In $k$-way merges, it may help to always consult shortest inverted list first.

This is a heuristic and might miss some top-$n$ results!
Threshold Methods: MaxScore

Top-$n$ formulation returns only documents with $score \geq \tau$.

→ But we know $\tau$ only after we evaluated the query!

However:

■ Once we added $n$ elements to the priority queue $R$, we can conclude that

$$\tau \geq \tau' \overset{\text{def}}{=} \text{minimum score in } R.$$  

i.e., $\tau'$ is a conservative estimate for $\tau$.

■ For each inverted list $l_j$, maintain maximum score $\mu_j$.

→ Once $\tau' > \mu_j$, documents that occur only in $l_j$ can be skipped.

MaxScore achieves similar effect as conjunctive processing, but guarantees a correct result.
We assumed that posting lists are *sorted by document id*.  
→ Enables delta encoding.  
→ Eases intersection/merging.  

Document ids, however, were so far assigned “randomly”.

**Idea:**  
- Assign document ids/order inverted lists, so list processing can be **terminated early**.  
- *E.g.*, order by **decreasing value of quality features**.  
  → $\mu_j$ decreases within $l_j$.  

© Jens Teubner · Architecture & Implementation of DBMS · Summer 2018
Inverted Lists with More Details

So far:
- Inverted lists contain document ids (pointers to documents).
- Must read (maybe even parse, tokenize, stem) documents to get $q_{ij}$.

Instead:
- Add information to inverted lists to **avoid document access**.
- Example: Add
  - number of documents that contain the term ($\sim idf_j$)
  - number of occurrences of the term in the document ($\sim tf_{ij}$)

<table>
<thead>
<tr>
<th>term</th>
<th>#</th>
<th>docs</th>
<th>term</th>
<th>#</th>
<th>docs</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>1</td>
<td>\langle doc_1:1 \rangle</td>
<td>both</td>
<td>1</td>
<td>\langle doc_1:1 \rangle</td>
</tr>
<tr>
<td>aquarium</td>
<td>1</td>
<td>\langle doc_3:1 \rangle</td>
<td>bright</td>
<td>1</td>
<td>\langle doc_3:1 \rangle</td>
</tr>
<tr>
<td>are</td>
<td>2</td>
<td>\langle doc_3:1, doc_4:1 \rangle</td>
<td>coloration</td>
<td>2</td>
<td>\langle doc_3:1, doc_4:1 \rangle</td>
</tr>
<tr>
<td>around</td>
<td>1</td>
<td>\langle doc_1:1 \rangle</td>
<td>derives</td>
<td>1</td>
<td>\langle doc_4:1 \rangle</td>
</tr>
<tr>
<td>as</td>
<td>1</td>
<td>\langle doc_2:1 \rangle</td>
<td>due</td>
<td>1</td>
<td>\langle doc_3:1 \rangle</td>
</tr>
</tbody>
</table>
Instead, some systems store **word positions**:

<table>
<thead>
<tr>
<th>term</th>
<th>#</th>
<th>docs</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>1</td>
<td>⟨doc₁:(15)⟩</td>
</tr>
<tr>
<td>aquarium</td>
<td>1</td>
<td>⟨doc₃:(5)⟩</td>
</tr>
<tr>
<td>are</td>
<td>2</td>
<td>⟨doc₃:(3), doc₄:(14)⟩</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>fish</td>
<td>4</td>
<td>⟨doc₁:(2, 4), doc₂:(7, 18, 23), doc₃:(2, 6), doc₄:(3, 13)⟩</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

→ Find phrases ("tropical fish") or rank documents higher where search terms occur nearby.
Inverted Lists with More Details

Store $tf_{ij}idf_i$ directly in inverted list?

✔ **Speeds up** computation of document scores.
   → Could incorporate even more expensive offline computations.

✘ **Very inflexible.**
   → What if ranking function changes? Need to re-compute index!

✘ Scoring values might **compress** poorly.

More Tricks:

- Store **extent lists** as inverted lists:
  → *E.g.*, inverted list for ‘title’, storing **document regions** that correspond to the document’s title.
  → Fits well with start/end tags in markup languages.
Evaluating a Search Engine

A good search engine returns
- many relevant documents, but
- few non-relevant documents.

“Relevant”? 
- What matters is relevance to the user.
- To evaluate a search engine
  → Take a test collection of documents and queries.
  → Obtain relevance judgements from experts (users).
  → Compare search engine output to expert judgements.
Recall and Precision

Recall:
- How many of the relevant documents were retrieved?

\[
Recall = \frac{|\text{retrieved documents that are relevant}|}{|\text{all relevant documents}|}
\]

Precision:
- How many of the retrieved documents are relevant?

\[
Precision = \frac{|\text{retrieved documents that are relevant}|}{|\text{retrieved documents}|}
\]

Since we return top-\(n\) documents according to rank, both values will vary with \(n\).
Recall and Precision

Precision and recall for an example document/query:

- Relevant Documents

- Result Document

© Jens Teubner · Architecture & Implementation of DBMS · Summer 2018
Recall and Precision

- Recall is **monotonically increasing**.
- Precision tends to **decrease** with \( n \).

→ Draw “recall-precision graph”