Architecture and Implementation of Database Systems (Summer 2018)

Jens Teubner, DBIS Group
jens.teubner@cs.tu-dortmund.de

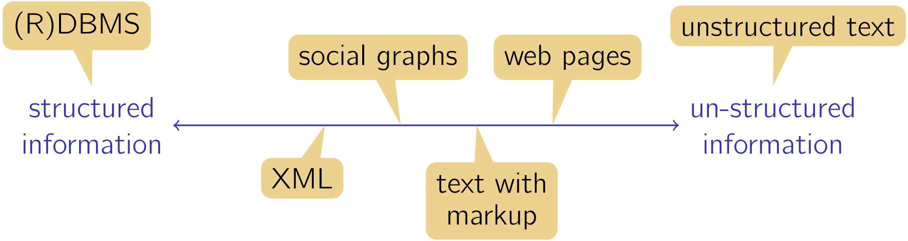
Summer 2018

Part XI

Search

Search

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



- How can we efficiently store and access such **un-structured data**?
  $\rightarrow$ success of **search engines** $\leadsto$ **“search”**

Boolean Queries

Let's start with what we have...

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

doc$_1$	doc$_2$	doc$_3$	doc$_4$
Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.	Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with salt-water tropical fish referred to as marine fish.	Tropical fish are popular aquarium fish, due to their often bright coloration.	In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

- Say we're interested in "freshwater fish."

  $\rightarrow$ Two **search terms**: "freshwater" and "fish"

Boolean Queries

Query in SQL-style notation:

```
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:

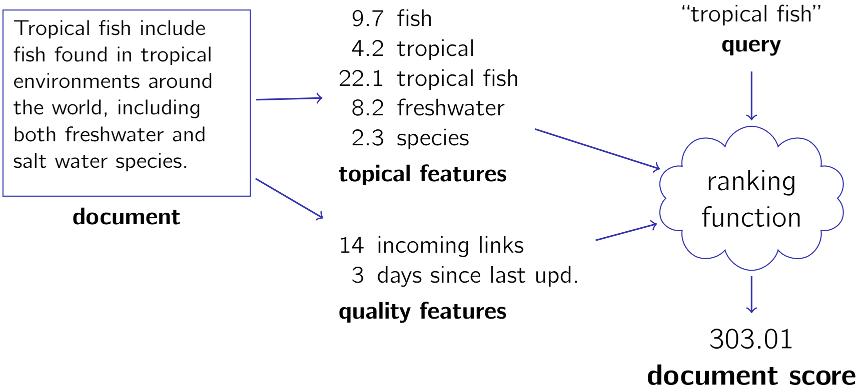
- Results are **ranked** according to their **relevance** (to the query).

Ranking

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**.



Ranking

Idea:

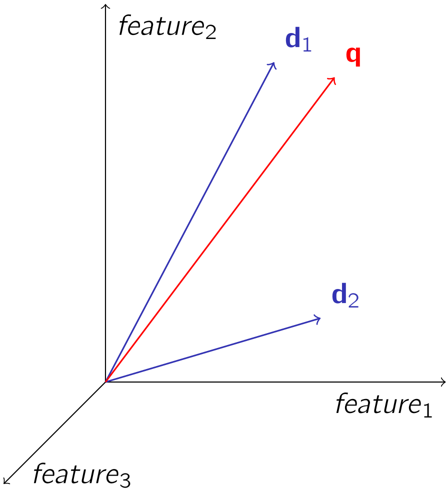
- Compute **similarity** between query and document.

Similarity:

- Define a set of **features** to use for ranking.
  - $\rightarrow$ each **term** in the collection is one feature
  - $\rightarrow$ possible features: document size/age, page rank, etc.
- For each **document** compute a **feature vector** $\mathbf{d}_i$
  - $\rightarrow$ *e.g.*, yes/no features; term count; etc.
- For the **query** compute a **feature vector** $\mathbf{q}$.
- Measure **similarity** of the two vectors.

Vector Space Model

Two vectors are similar if the **angle** between them is small.



Cosine between $\mathbf{d}_i$ and $\mathbf{q}$:

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

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

→ "vector space model"

Ranking Model

Ignoring the normalization term: $\mathrm{sim}(\mathbf{d}_i, \mathbf{q}) = \sum_j d_{ij} q_j$.

$\rightarrow$ 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.

document

9.7	fish	$\longleftrightarrow$	fish	5.2
4.2	tropical	$\longleftrightarrow$	tropical	3.4
22.1	tropical fish	$\longleftrightarrow$	tropical fish	9.9
8.2	freshwater		chichlids	1.2
2.3	species		barbs	0.7
topical features			**topical features**	
14	incoming links	$\longleftrightarrow$	incoming links	1.2
3	days last upd.	$\longleftrightarrow$	days last upd.	0.9
quality features			**quality features**	

"tropical fish"
query

303.01
document score

✎ **What does this mean for an implementation?**

tf / *idf* Ranking

What are good features (and their values)?

Topical Features:

- Each **term** in the collection ($\leadsto$ vocabulary) is one feature.

Feature Value:

- A document with **multiple occurrences** of 'foo' is likely more relevant to queries that contain 'foo'.

  $\rightarrow$ **term frequency** *tf* as a feature value.

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

$\rightarrow$ Normalize to account for different document sizes.

tf / *idf* Ranking

- Terms that occur **in many documents** are less discriminating.

  → **inverse document frequency** *idf*:

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

  → *idf* is a property of the **term**, not the document!

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

$$d_{ij} = tf_{ij} \cdot idf_j \quad .$$

- Do the same thing for **query** features q_j.

tf / idf Ranking

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

PageRank[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.

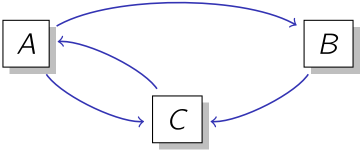
Idea:

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

[28]Named after Google founder Larry Page.

Computing PageRank

Example:



Probability that Alice ends up on C:

$$PR(C) = \underbrace{\frac{\lambda}{3}}_{\text{random jump}} + \underbrace{(1-\lambda)\cdot\left(\frac{PR(A)}{2}+\frac{PR(B)}{1}\right)}_{\text{chance of coming from } A \text{ or } B} \;.$$

Generally:

$$PR(u) = \frac{\lambda}{N} + (1-\lambda)\cdot\sum_{v\in B_u}\frac{PR(v)}{outgoing_v} \;.$$

Computing PageRank

But we don't know $PR(A)$ and $PR(B)$, yet!

$\rightarrow$ **Iterate** the above formula and PageRanks will converge.

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

- A typical value for λ is 0.15.
- Today, PageRank is just one out of many features used in ranking.
  - $\rightarrow$ 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

$$\textit{document} \rightarrow \text{list of } \textit{term} \quad .$$

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

$$\textit{term} \rightarrow \text{list of } \textit{document} \quad .$$

E.g.,

term	*docs*
and	(doc_1)
aquarium	(doc_3)
are	(doc_3, doc_4)
around	(doc_1)
as	(doc_2)

term	*docs*
both	(doc_1)
bright	(doc_3)
coloration	(doc_3, doc_4)
derives	(doc_4)
due	(doc_3)

Inverted Files

A representation of this type is thus also called **inverted file**[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**.

[29] sometimes also "inverted index"

Size of Inverted Files

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.

✎ **How well will lists of document ids compress?**

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

$$1, 5, 9, 18, 23, 24, 30, 44, 45, 48$$

$$\rightsquigarrow$$

$$1, 4, 4, 9, 5, 1, 6, 14, 1, 3$$

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 → $\underline{00}$ 00010010; 427 → $\underline{01}$ 00000001 10101011.

→ *E.g.*, **unary codes**
- Encode n with sequence of n 1s, followed by a 0.
- *E.g.*, 0 → 0; 1 → 10; 2 → 110; 12 → 1111111111110.

Elias-γ Codes:

- To encode n, compute

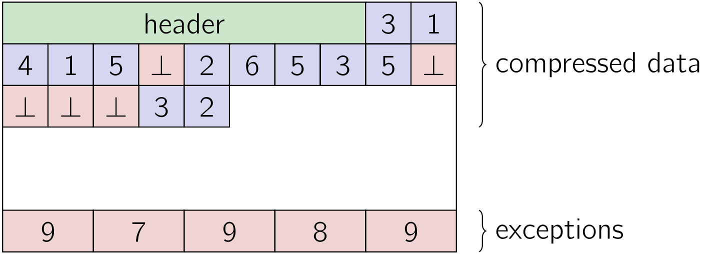
$$\begin{aligned} n_d &= \lfloor \log_2 n \rfloor && \text{“position of leading bit”} \\ n_r &= n - 2^{\lfloor \log_2 n \rfloor} && \text{“value encoded by remaining bits”} \end{aligned}$$

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

n	n_d	n_r	code
1	0	0	0
2	1	0	10 0
3	1	1	10 1
15	3	7	1110 111
255	7	127	11111110 1111111

PFOR Compression:

- Illustrated here using compressed representation of the digits of π.[30]



- Decompressed numbers: 31415926535897932

[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:

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

For PFOR, `DECODE` is a simple addition:

```
#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.

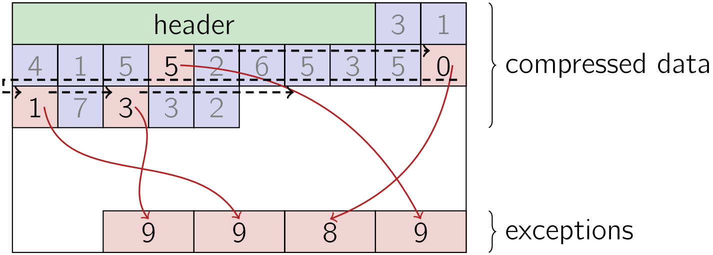
```
/* ignore exceptions during decompression */
for (i = 0; i < n; i++)
    output[i] = DECODE (code[i]);

/* patch the result */
foreach exception
   patch corresponding output item ;
```

PFOR: Patching the Output

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.

PFOR: Patching the Output

The resulting decompression routine is branch-free:

```
/* 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];
}
```

Query Execution—Boolean Queries

With inverted lists available, the evaluation of

$$term_1 \;\texttt{and}\; term_2$$

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

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

→ "Merge" lists I_{term_1} and I_{term_2} (↗ `merge_join()`, slide 186).

→ Cost: linear scan of I_{term_1} plus linear scan of I_{term_2}.

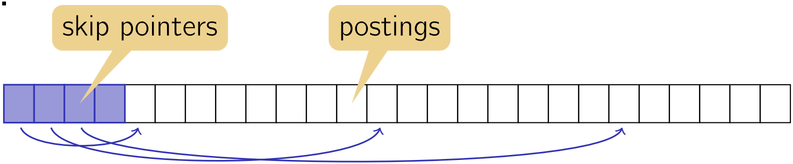
Problem: Long, inefficient scans

E.g.,

- $|I_{fish}| = 300\,\text{M}$; $|I_{freshwater}| = 1\,\text{M}$.
- At least 299 M I_{fish} entries scanned unnecessarily.
  → **Skip** over those entries?

Skip Pointers

Idea:



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

Skip forward to document d:

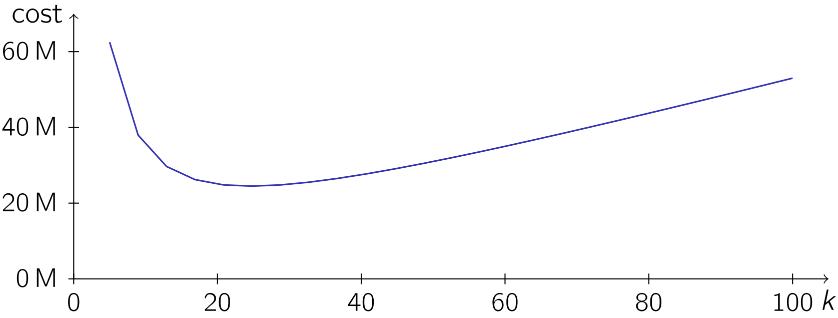
1. Read skip pointer list as long as $\mathit{doc\ id} \leq d$.
2. Follow the pointer and scan posting list from there to find d.

Skip Pointers

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

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

- Read all $300\,\text{M}/k$ skip pointers.
- Perform $1\,\text{M}$ 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 $\sum_j d_{ij} q_j$.

$\rightarrow$ Score only documents that appear in at least one inverted list for the index terms in $\mathbf{q}$.

Term-at-a-Time Retrieval

Process inverted lists one after another:

```
R ← PriorityQueue (n) ;
A ← HashTable () ;
foreach term j in q do
    foreach document i in inverted list for j do
        score ← A.get (i) ;
        if not found then
            A.put (i, d_ij q_j) ;
        else
            A.put (i, score + d_ij q_j) ;
foreach ⟨i, score⟩ in A do
    R.add (i, score) ;
return R ;
```

Document-at-a-Time Retrieval

```
R ← PriorityQueue(n) ;
foreach term j in q do
    L.add(inverted list for j) ;
while L is not empty do
    /* Find next document i in any inverted list            */
    i ← smallest l_j.docID in L ;
    /* Score document i                                      */
    score ← 0 ;
    foreach l_j ∈ L do
        if l_j.docID = i then
            score ← score + d_ij q_j ;
            l_j.advance() ;
            if eof(l_j) then
                L.remove(l_j) ;
    R.add(i, score) ;
return R ;
```

Optimizations: Conjunctive Processing

Restriction:

- Return only documents that contain **all** of the query terms.

Then:

- Document-at-a-time $\leadsto$ 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 $score \geq \tau$.

$\rightarrow$ But we know τ only after we evaluated the query!

However:

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

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

  i.e., τ' is a conservative estimate for τ.

- For each inverted list l_j, maintain **maximum score** μ_j.

  $\rightarrow$ 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**.

List Ordering

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**.
  → μ_j decreases within I_j.

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 ($\rightsquigarrow$ idf_j)
  - number of occurrences of the term in the document ($\rightsquigarrow$ tf_{ij})

term	#	*docs*
and	1	$(\langle doc_1{:}1\rangle)$
aquarium	1	$(\langle doc_3{:}1\rangle)$
are	2	$(\langle doc_3{:}1\rangle, \langle doc_4{:}1\rangle)$
around	1	$(\langle doc_1{:}1\rangle)$
as	1	$(\langle doc_2{:}1\rangle)$

term	#	*docs*
both	1	$(\langle doc_1{:}1\rangle)$
bright	1	$(\langle doc_3{:}1\rangle)$
coloration	2	$(\langle doc_3{:}1\rangle, \langle doc_4{:}1\rangle)$
derives	1	$(\langle doc_4{:}1\rangle)$
due	1	$(\langle doc_3{:}1\rangle)$

Inverted Lists with More Details

Instead, some systems store **word positions**:

term	#	*docs*
and	1	$(\langle doc_1\colon (15)\rangle)$
aquarium	1	$(\langle doc_3\colon (5)\rangle)$
are	2	$(\langle doc_3\colon (3)\rangle, \langle doc_4\colon (14)\rangle)$
⋮	⋮	⋮
fish	4	$(\langle doc_1\colon (2,4)\rangle, \langle doc_2\colon (7,18,23)\rangle,$ $\langle doc_3\colon (2,6)\rangle, \langle doc_4\colon (3,13)\rangle)$
⋮	⋮	⋮

→ 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 engines 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:

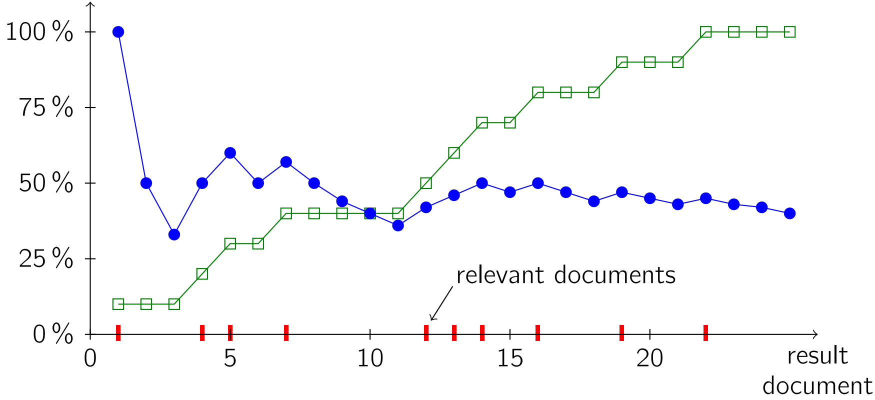
- How many of the retrieved documents are relevant?

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

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:



Recall and Precision

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

→ Draw "recall-precision graph"