



IN4325

Indexing and query processing

Claudia Hauff (WIS, TU Delft)

The big picture

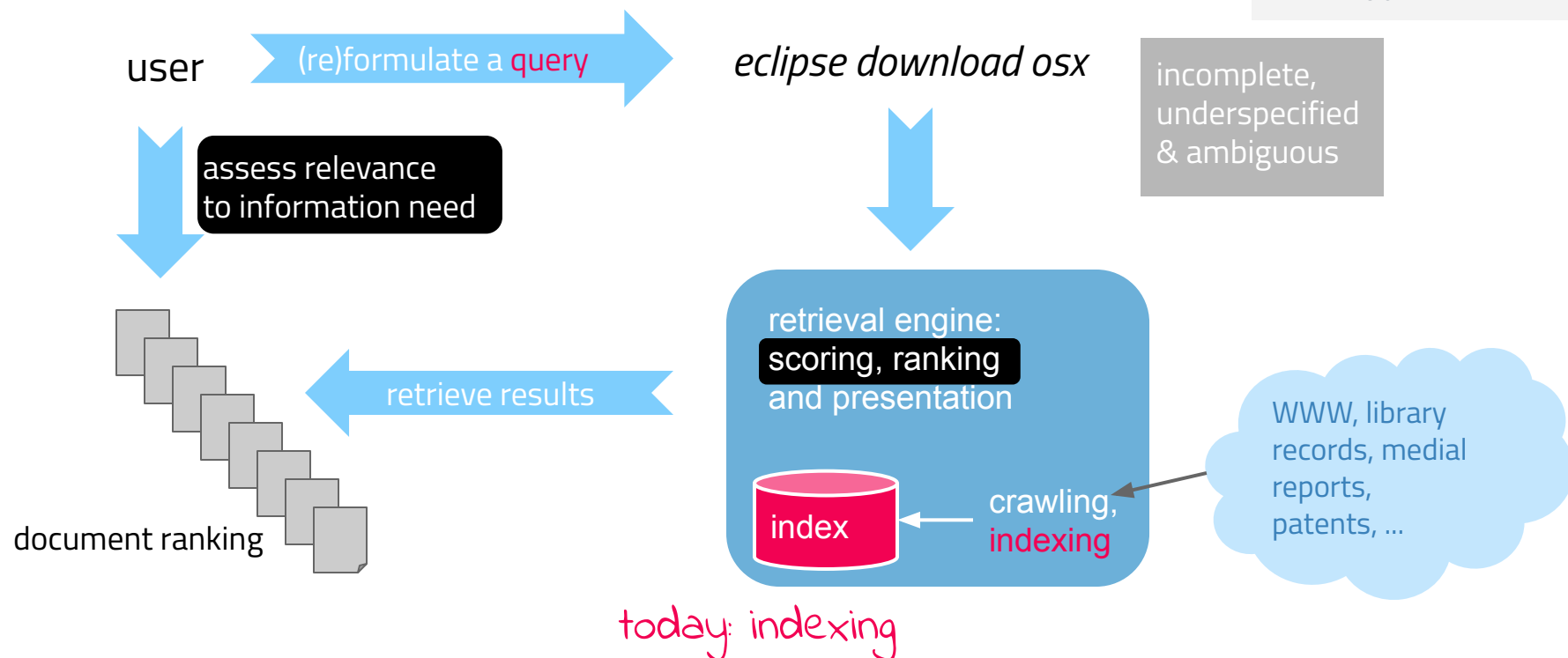
The essence of IR

Information need: *Looks like I need Eclipse for this job. Where can I download the latest beta version for macOS Sierra?*

Information need
Topic the user wants to know more about

Query
Translation of need into an input for the search engine

Relevance
A document is relevant if it (partially) provides answers to the information need



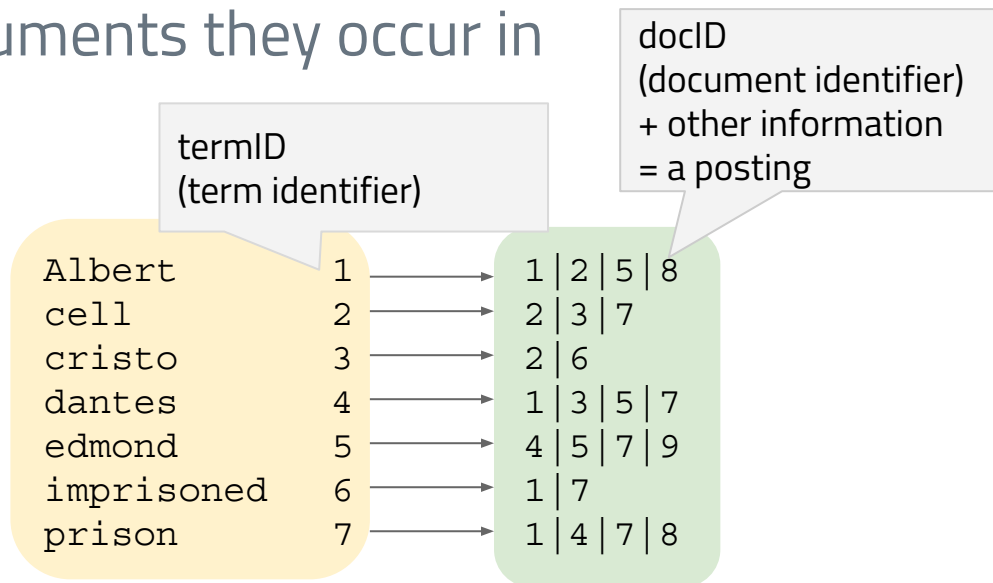
Terminology

Relatively easy in English
(majority of docs on the Web).
Less trivial in other languages
or **mixed script** documents.

Inverted index maps terms back to the part of the documents they occur in



What's the
issue with a
file-based
posting list?



dictionary
(entries sorted
alphabetically);
relatively small

postings lists
(postings often
ordered by docIDs);
huge

Often, terms==normalized
tokens. Not required though.

What is a **document
unit** depends on the
application.

- 1) Collect the documents to index

"I am not going there to be
imprisoned," said Dantes. D1

- 2) Tokenize the content (from string
to tokens)

"I am not going there to be
imprisoned," said Dantes.

- 3) Normalize the tokens
(preprocessing), decide on terms

i am not go there to be imprison
said dantes
(imprison, D1) pair

- 4) Index the documents

Inverted index

- The computational equivalent of the index at the back of most textbooks
- Basic position information and pointers
- “Inverted”: usually words are part of *documents*, now documents belong to words

absolute error, 437
accuracy, 359
ad hoc search, 3, 280, 423
adaptive filtering, 425
adversarial information retrieval, 294
advertising, 218, 371
 classifying, 371
 contextual, 218–221
agglomerative clustering, 375
anchor text, 21, 56, 105, 280
API, 439, 461
architecture, 13–28
authority, 21, 111
automatic indexing, 400

background probability, *see* collection probability
bag of words, 345, 451
Bayes classifier, 245
Bayes Decision Rule, 245
Bayes’ Rule, 246, 343
Bayes’ rule, 342
Bayesian network, 268
bibliometrics, 120
bidding, 218
bigram, 100, 253
BigTable, 57

binary independence model,
blog, 111
BM25, 250–252
BM25F, 294
Boolean query, 235
Boolean query language, 24
Boolean retrieval, 235–237
boosting, 448
BPREF, 322
brute force, 331
burstiness, 254

caching, 26, 181
card catalog, 400
case folding, 87
case normalization, 87
categorization, *see* classification
CBIR, *see* content-based image retrieval
character encoding, 50, 119
checksum, 60
Chi-squared measure, 202
CJK (Chinese-Japanese-Korean)
classification, 3, 339–373
 faceted, 224
 monothetic, 223, 374
 polythetic, 223, 374
classifier, 21

Umbrella term for different data structures

Inverted index

Suitable data structures depend on the retrieval models employed.



Indexing as an
offline process



Wide variety of
retrieval models
(direct access to
index data structure)

PUT /index

```
{
  "settings": {
    "number_of_shards": 1,
    "similarity": {
      "scripted_tfidf": {
        "type": "scripted",
        "script": {
          "source": "double tf = Math.sqrt(doc.freq);
        }
      }
    },
  },
  "mappings": {
    "_doc": {
      "properties": {
        "field": {
          "type": "text",
          "similarity": "scripted_tfidf"
        }
      }
    }
  }
}
```



elastic

Retrieval model
required for
index creation
(low level details
remain hidden)

2.8 billion Web pages

240 TB uncompressed
content

850 million new URLs 01/2019

はい C'est vrai! RAW DATA
Tak! 40+ languages Si. METADATA
Efectivamente. हाँ TEXT DATA
You bet.

7
YEARS OF DATA

Common Crawl



Academic corpora

WT10g: 1.7 million documents

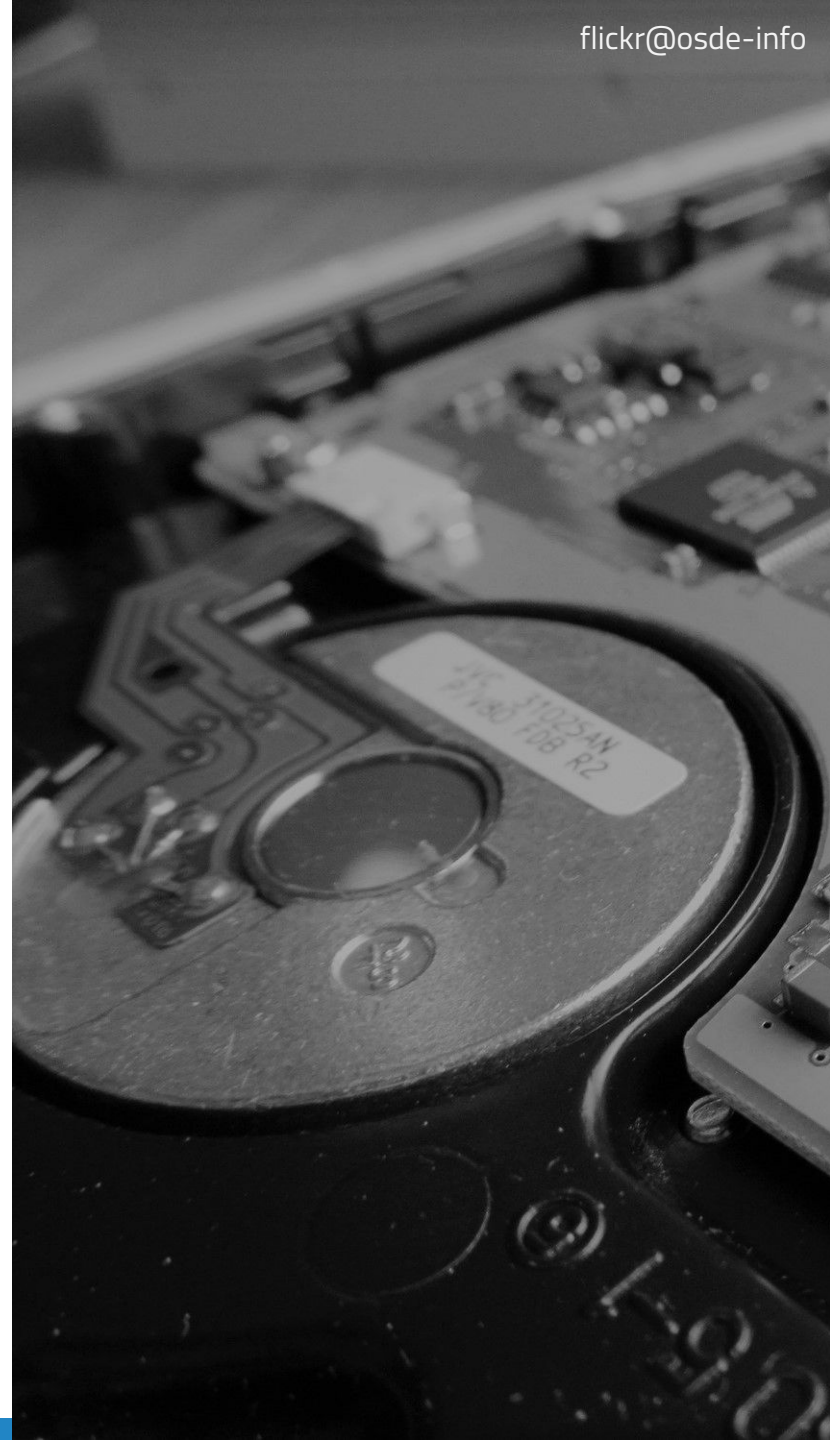
GOV2: 25.2 million documents



*Choosing the optimal encoding for an inverted index is an **ever-changing game** for the system builder, because it is strongly dependent on underlying computer technologies and their **relative speed and sizes**.*

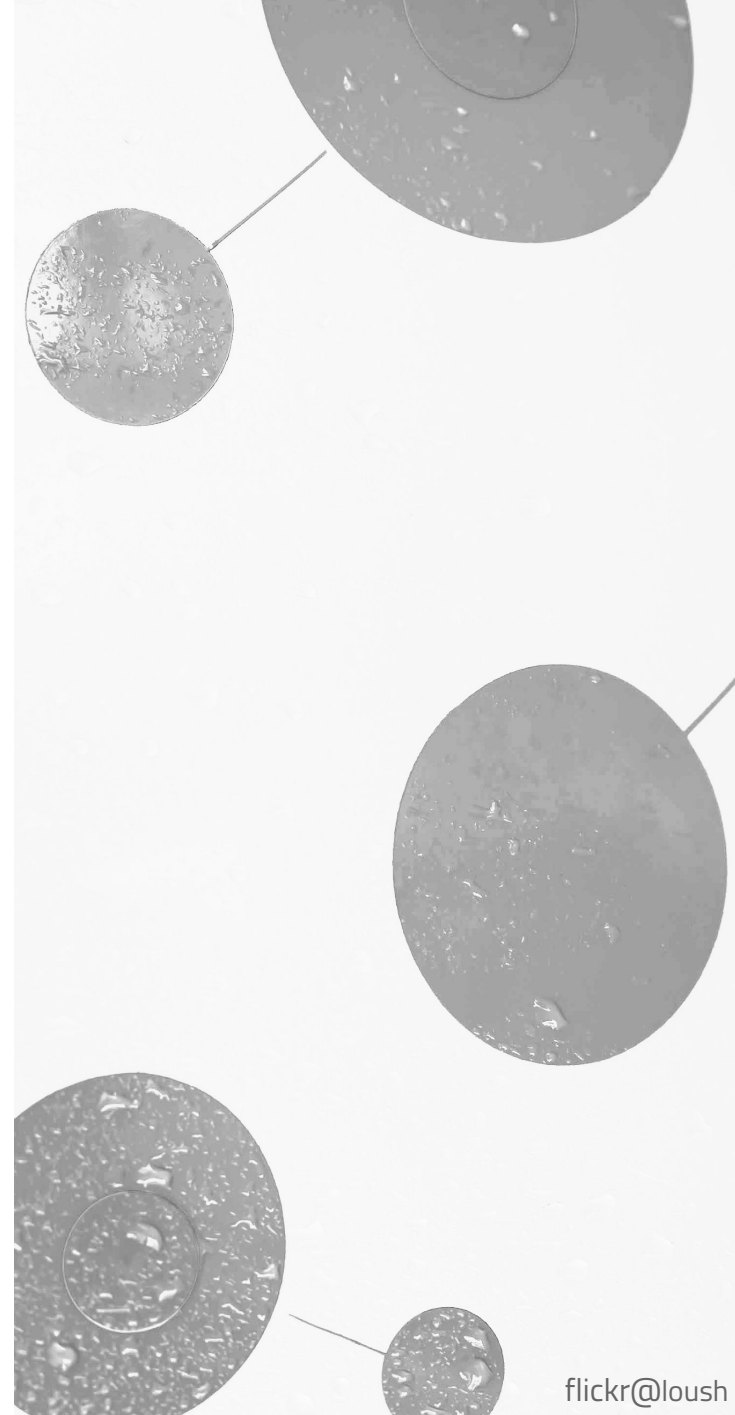
Hardware constraints to think about

- **Disks** maximize input/output throughput if contiguously stored data is accessed
- **Memory** access is faster than disk access
- An OS **reads/writes blocks** of fixed size from/to disk
- Reading **compressed** data + decompressing is faster than reading uncompressed data from disk



To come in this lecture

1. Structure & types of inverted indices (boolean vs. ranked retrieval)
2. Compression algorithms
3. Index construction
4. Query processing
5. Distributed indexing



Boolean retrieval: appropriate index structures

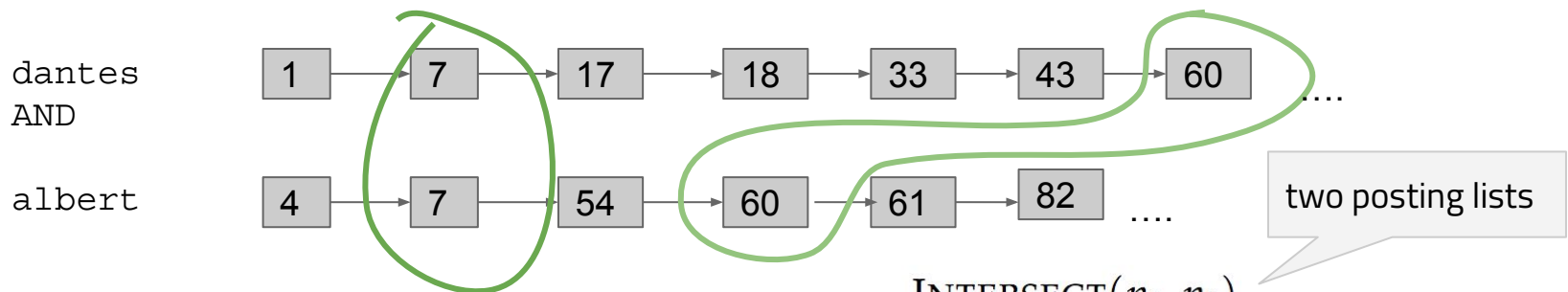
Is this really complicated?

- Searching for the lines in the book *Count of Monte Christo* that contain the terms **Dantes** AND **prison** but NOT **Albert**
- Naive solution:

```
more infile |grep Dantes|grep prison|grep -v Albert
```
- Problems:
 - **Proximity operators** not easy to implement, e.g. **Dantes** within at most 3 terms of **prison**
 - **Approximate/semantic matches** require users to think ahead, e.g. (**Edmond** OR **Dantes**) AND (**prison** OR **cell** OR **imprisoned**) NOT **Albert**

Boolean retrieval in one slide

Dantes AND Albert



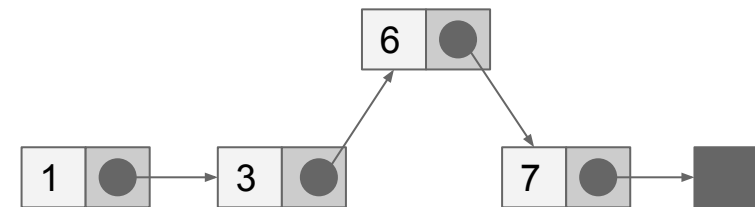
- 1) **Preprocess the query in the same manner as the corpus**
- 2) Determine whether both query terms exist
- 3) Locate pointers to the respective postings lists
- 4) Intersect posting lists

```
INTERSECT( $p_1, p_2$ )
1   $answer \leftarrow \langle \rangle$ 
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3  do if  $\text{docID}(p_1) = \text{docID}(p_2)$ 
4      then  $\text{ADD}(answer, \text{docID}(p_1))$ 
5           $p_1 \leftarrow \text{next}(p_1)$ 
6           $p_2 \leftarrow \text{next}(p_2)$ 
7      else if  $\text{docID}(p_1) < \text{docID}(p_2)$ 
8          then  $p_1 \leftarrow \text{next}(p_1)$ 
9          else  $p_2 \leftarrow \text{next}(p_2)$ 
10 return  $answer$ 
```

Postings lists

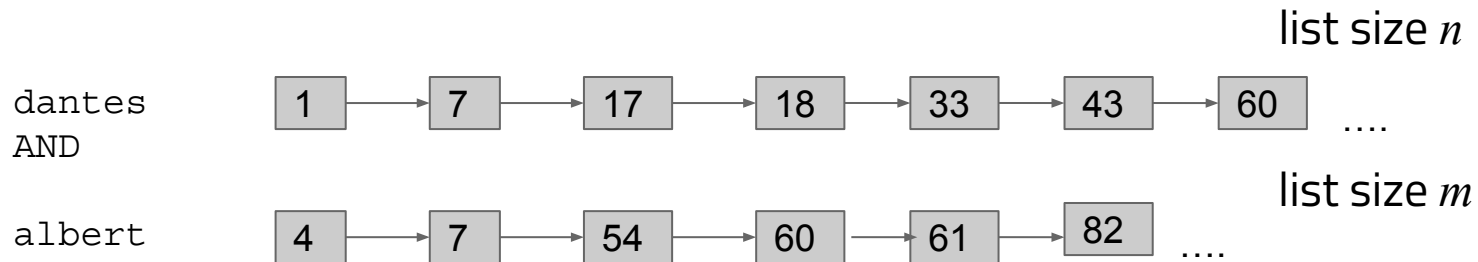
Postings lists: data structures

- Index needs to be optimized for storage and access efficiency
- Different implementations of postings lists with different **advantages** and **disadvantages**:
 1. Fixed length array: **easy**, **wastes a lot of space**
 2. Singly linked list: **cheap insertion**
 3. Variable length arrays
 - a. **Require less space** than linked lists (no pointers)
 - b. **Allow faster access** (contiguous memory increases)
 - c. Good if **few updates** are required



Skip pointers

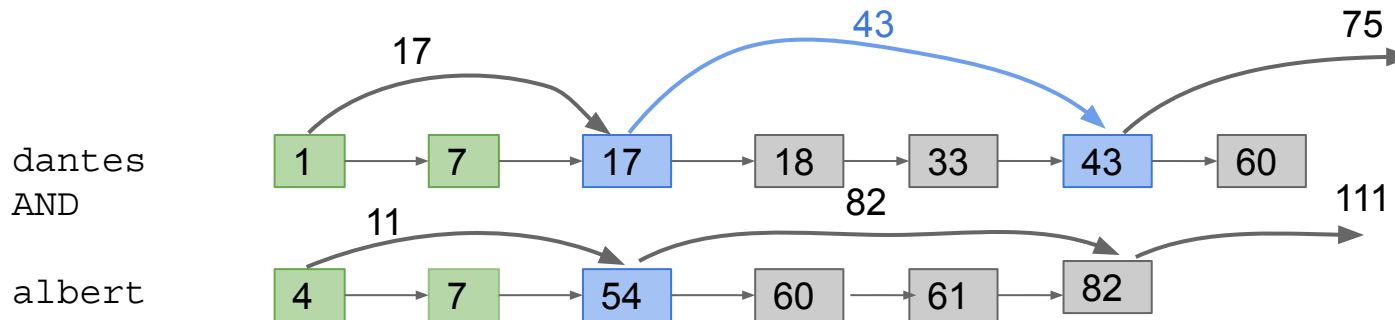
Skip pointers (created at indexing time)



List intersection without skip pointers: $O(n+m)$

Skip pointers

Skip pointers are shortcuts



List intersection without skip pointers: $O(n+m)$

List intersection with skip pointers: sublinear



What about OR queries?

Skip pointers: where to place them

- **Tradeoffs:**

- More skips yield shorter skip spans; more skips are likely (many skip pointer comparisons & pointer storage)
- Fewer skips yield larger skip spans; few skips are likely (requires few comparisons, less space)

- **Heuristic:** for postings lists of length L , use \sqrt{L} evenly spaced skip pointers

- Effective skip pointers are relatively easy to create in static indices, harder when the postings lists are frequently updated (*also depends on hardware: the entire postings list has to be read from disk*)

On the issue of multi-word compounds

Multi-word compounds

- Concepts and names may be **multi-word compounds**, e.g. "Edmond Dantes"
 - If treated as a phrase, it should not return the sentence "Edmond went to the town of Dantes."
 - Web search engines introduced the "..." syntax for phrase queries (~10% of queries are phrase queries)
- Postings lists of the form $termID \rightarrow d1|d2|d3|...$ do not provide sufficient granularity
 - Require substantial **post-retrieval filtering**

Beyond the basic index



What can we do if we also want to include phrases like "declaration of independence"?

Biword indices

biword: a pair of consecutive words

I am not going there to be imprisoned ...

i am

am not

not going

going there

there to

to be

be imprisoned

- Each biword is one **vocabulary term**
- Two-word phrase queries can be handled immediately
- Longer phrase queries are broken down, e.g. "Count of Monte Cristo" becomes "Count of" AND "of Monte" AND "Monte Cristo" (false positives possible)

Biword indices

- Can be extended to longer and variable length sequences (phrase indices)
- Single term queries are not handled naturally in biword indices (entire index scan is necessary); additional single term index needed
- Arbitrary phrases are usually not indexed, **vocabulary sizes** increase greatly

The Count of Monte Cristo
~50K lines of text

<i>Vocabulary size</i>	
Single-term index	19,236
Biword index	866,914
Triword index	6,425,444



Positional indices: format

Most common index type

For each term, postings are stored with frequency values

to, 993427:

to occurs 993,427 times in the corpus

$\langle 1, 6: \langle 7, 18, 33, 72, 86, 231 \rangle;$

to occurs 6
times in
document 1

$2, 5: \langle 1, 17, 74, 222, 255 \rangle;$

to occurs at positions 7,
18, 33, 72, 86 and 231 in
document 1.

$4, 5: \langle 8, 16, 190, 429, 433 \rangle;$

$5, 2: \langle 363, 367 \rangle;$

$7, 3: \langle 13, 23, 191 \rangle; \dots \rangle$

be, 178239:

$\langle 1, 2: \langle 17, 25 \rangle;$

$4, 5: \langle 17, 191, 291, 430, 434 \rangle;$

$5, 3: \langle 14, 19, 101 \rangle; \dots \rangle$

Positional indices: querying

- To process a phrase query: "to be or not to be"
 1. Access the postings list for each term
 2. When merging (intersecting) the result list, check if the positions of the terms match the phrase query
 - a. Calculate offset between terms
 - b. Start with the least frequent term
- Increased index size: the index is 2-4x larger than a non-positional index
- In practice: combine biword and positional indices

The dictionary

Dictionary lookup

- Also known as *lexicon* or *vocabulary*
 - 1) Determine whether all query terms exist
 - 2) Locate pointers to the respective posting lists
- Implementation options are **hashes** or **search trees** depending on:
 - Number of terms (keys)
 - Frequency and type of index changes (key insert/delete)
 - Frequency of key accesses

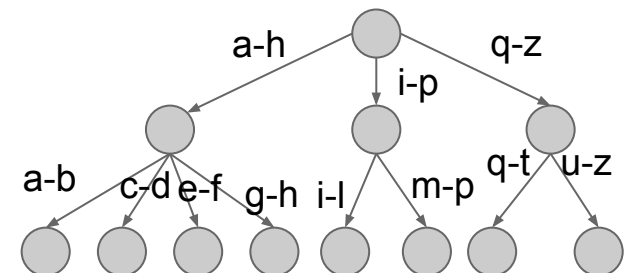
Hashes

Idea: each *vocabulary term* is hashed into an integer

- Querying: hash each term separately, follow pointer to corresponding postings list
- Disadvantage I: unable to react to slight differences in query terms (e.g. *Dantes* vs. *Dante*)
- Disadvantage II: unable to seek for all terms with a particular prefix (e.g. *Dant**)

Search trees

- **Binary search trees** overcome the two disadvantages of the hash implementation
- Care needs to be taken when terms are added/deleted from the tree (might require rebalancing)
- In practice: **B-trees** is the data structure of choice (self-balancing search tree with #children in $[a,b]$)



Wildcard queries

Commonly employed when:

- There is **uncertainty** about the spelling of a term
- Multiple **spelling variants** of a term exist (labour vs labor)
- All terms with the same **stem** are wanted (restoration vs restore)

Trailing wildcard query: `restor*`

- Dictionary via as search tree: walk along the edges and enumerate the W terms with prefix `restor`; followed by $|W|$ lookups of the respective postings lists to retrieve all docIDs

Wildcard queries

Leading wildcard query: **building* (*building* vs. *rebuilding*)

- Reverse dictionary B-tree: constructed by reading each term in the vocabulary backwards
- Reverse B-tree is traversed backwards: g-n-i-d-l-i-u-b

Single wildcard query: *analy*ed* (*analyzed* vs *analysed*)

- Traverse the regular B-tree to find the W terms with prefix *analy*
- Traverse the reverse B-tree to find the R terms with suffix *ed*
- Final result: intersect W and R

Beyond boolean retrieval

A high-level view

Feature: any attribute we can express numerically

topical features
(query-independent)

term
term
term
...
category
...

query

document
score

Ranking
function

PageRank
domain
readability
#hyperlinks
last update
...

feature function

$$RSV(Q, D) = \sum_i g_i(Q) \times f_i(D)$$

feature i

quality features
(query-independent)



WIKIPEDIA

The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages

Article Talk

Read

Edit

View history

Search Wikipedia

Await

From Wikipedia, the free encyclopedia

For a definition of the word "await", see the Wiktionary entry await.
In computer programming, **await** is a feature found in C# 5.0, Python 3.5, Hack, Dart, Kotlin 1.1, in an experimental extension for Scala,^[1] and more recently JavaScript that allows an asynchronous, non-blocking method call to be performed in a similar way to an ordinary synchronous method call.
While a casual reading of the code would suggest that the method call blocks until the requested data is available, in fact it does not.

Contents [hide]

- 1 Basic Operation
- 2 In F#
- 3 In C#
- 4 In Scala
- 4.1 How it works
- 5 In Python
- 6 In JavaScript
- 7 See also
- 8 References

Basic Operation [edit]

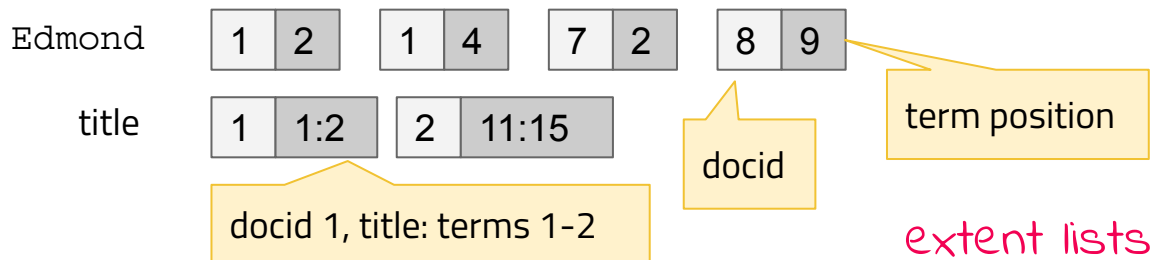
What `async` and `await` do are essentially perform compile-time readjustments of your code. They use the Task data structure to maintain the state of your computation.
Let's look at a small snippet of code that uses `await`:

```
public async Task<int> FindPageSize(Uri uri)
{
    var data = await new WebClient().DownloadDataTaskAsync(uri);
    return data.Length;
}
```

Complex retrieval models ...

Require additional information to be stored in the postings lists

- presence/absence of terms in documents
- term counts
- term positions
- document fields (e.g. header, title, footer) BM25F



A query with N terms in most cases requires the scan of N postings lists

How can we deal with semantic approaches?



Auxiliary data

Most retrieval models require **global corpus statistics**:

- Vocabulary size
- Number of documents
- Average document length
- ...

Lemur/Indri stores those statistics in an XML file (generated during index creation)



Actual **document content** is not stored in an inverted index – is that a problem?

- Not for ranking, but for snippet generation
- Additional system needed to link docids to (cached) documents

Compression



Corpus of web documents uncompressed: 100%

Inverted index with position information: 30%

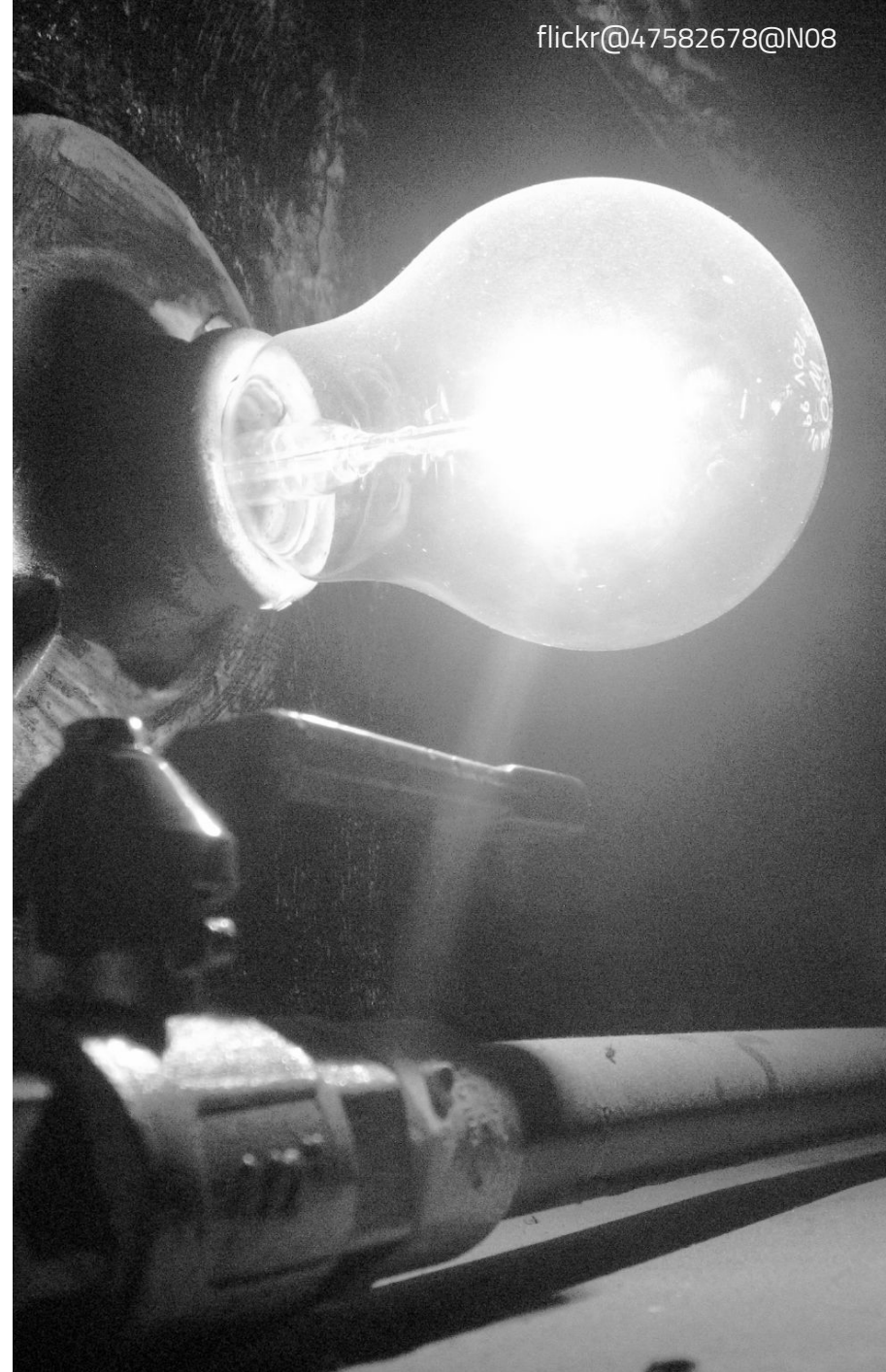
Inverted index with position information compressed: 10-15%

Overview

- **Memory hierarchy**: smallest and fastest (cache memory) vs. largest and slowest (disk)
- Compression aim: to make use of the hierarchy efficiently
- Inverted files of large collections are large themselves
- Compression enables:
 - **more data** can use fast levels of the memory hierarchy
 - to seek **more data** from disk at a time
- **Efficient** compression requires a fast decompression algorithm
- Text compression is **lossless** (in contrast to audio, video, ...)

Main insight

- Represent common terms (or termIDs, i.e. integers) with **short codes** and less frequent terms with **longer codes**
- Usage assumptions guide the way: e.g. word counts (**docids**) in postings lists tend (**not**) to be small



Delta encoding

Inverted file data mostly encoded as **positive integers** (document identifiers, term positions, ...)

If upper bound for x is known, x can be encoded in $\lceil \log_2 X \rceil$ *bits*

Inverted lists can be considered as a sequence of run length or **document gaps** between document numbers:

dantes 1 → 7 → 17 → 18 → 33 → 43 → 60

d-gaps 1 6 10 1 15 10 17

Delta encoding

Inverted file data mostly encoded as **positive integers** (document identifiers, term positions, ...)

If upper bound for x is known, x can be encoded in $\lceil \log_2 X \rceil$ *bits*

Inverted lists can be considered as a sequence of run length or **document gaps** between document numbers:

dantes 1 → 7 → 17 → 18 → 33 → 43 → 60

d-gaps 1 6 10 1 15 10 17

D-gaps are small for frequent terms, large for infrequent terms

Stopword	1, 1, 1, 2, 1, 1, 1, 3, 1, 1, 5, ...	(long list)
Rare term	74324, 432, 849503	(short list)

(Mostly) **small** integers can be compressed efficiently

Unary code

- Idea: use a **single symbol** to encode numbers

<i>Number</i>	<i>Symbol</i>
0	0
1	10
2	110
5	111110

why can't we just use binary code?

unambiguous decoding is not possible
101110101110110100

- Unary encoding is efficient for 0/1 but not 1023 (requires 10 bits in binary vs. 1024 in unary code)
- It is **unambiguous** & easy to decode

Elias-γ code

- Idea: combine the strength of unary and binary code
- To encode a number k we compute:

k_d is the number of binary digits needed to express k in binary form.

$$k_d = \lfloor \log_2 k \rfloor$$

$$k_r = k - 2^{\lfloor \log_2 k \rfloor}$$

unary code
= Elias-γ code

binary code

If $k > 0$ the leftmost digit is 1. Erase it. The remaining binary digits are k_r

space requirements (in bits)
for a number k :

$$2 \times \lfloor \log_2 k \rfloor + 1$$

k	k_d	k_r	Code
1	0	0	0
2	1	0	100
3	1	1	101
6	2	2	11001
15	3	7	1110111
1023	9	511	1111111110111111111

Refinement: Elias- δ code

- Elias- γ is not ideal for inputs that *may* contain large numbers
- Refinement: instead of encoding k_d in unary code (long for large numbers), encode it in Elias- γ code

Number (k)	k_d	k_r	k_{dd}	k_{dr}	Code
1	0	0	0	0	0
2	1	0	1	0	10 0 0
3	1	1	1	0	10 0 1
6	2	2	1	1	10 1 10
15	3	7	2	0	110 00 111
16	4	0	2	1	110 01 0000
255	7	127	3	0	1110 000 1111111
1023	9	511	3	2	1110 010 111111111

Elias- δ is less efficient for small numbers than Elias- γ but **more efficient for larger numbers.**

How does it all come together?

(1,1) (1,7) (2,6) (2,17) (2,197) (3,1) **posting list** (doc,position)

(1,2,[1,7]) (2,3,[6,17,197]) (3,1,[1]) **rewrite** (doc,count,[pos.])

brackets only
for readability

(1,2,[1,7]) (1,3,[6,17,197]) (1,1,[1]) **delta encoding** of docids

(1,2,[1,6]) (1,3,[6,11,180]) (1,1,[1]) **delta encoding** of positions

1 2 1 6 1 3 6 11 180 1 1 1

81 82 81 86 81 83 86 8B 01 B4 81 81 81 **v-byte** compression

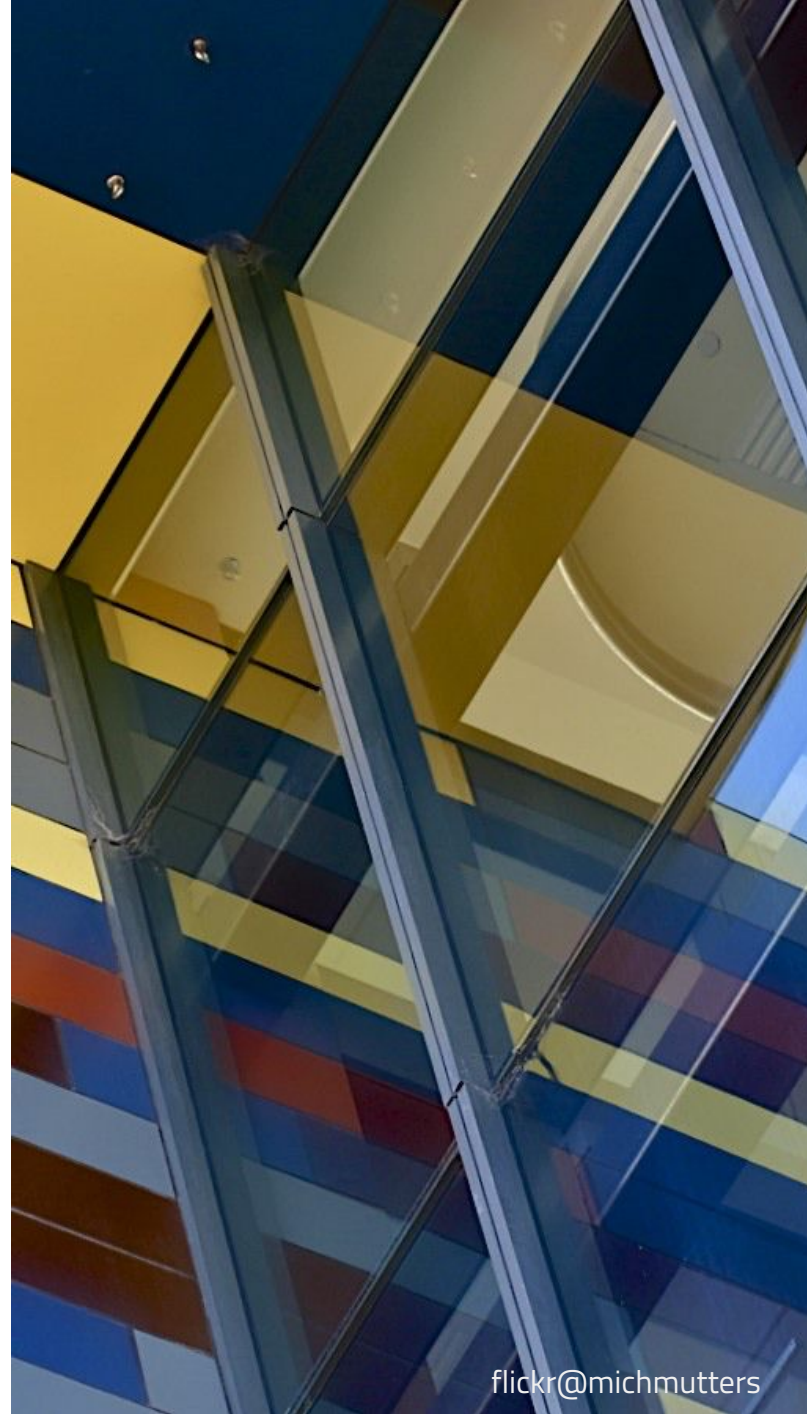
Earlier on we considered binary search (bs) within a posting list but this example shows that compression and bs are not easily compatible.

Index construction

How can we compute the inverted file when our document corpus has Terabytes or Petabytes of text?

Increasing complexity

- In-memory index construction
- Single machine (disk-based) index construction
- Cluster-based index construction (corpus does not fit onto a single machine)



In-memory indexing

procedure BUILDINDEX(D)

$I \leftarrow \text{HashTable}()$

$n \leftarrow 0$

All posting lists are
maintained in memory

▷ D is a set of text documents

▷ Inverted list storage

▷ Document numbering

for all documents $d \in D$ **do**

$n \leftarrow n + 1$

$T \leftarrow \text{Parse}(d)$

▷ Parse document into tokens

Remove duplicates from T

for all tokens $t \in T$ **do**

if $I_t \notin I$ **then**

$I_t \leftarrow \text{Array}()$

end if

$I_t.\text{append}(n)$

Requires additional
effort to parallelize

end for

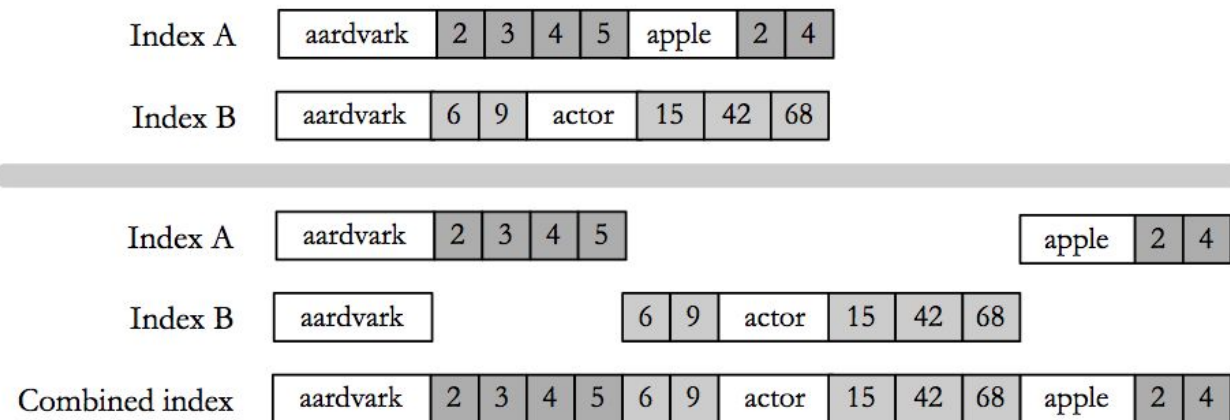
end for

return I

end procedure

Using the disk ...

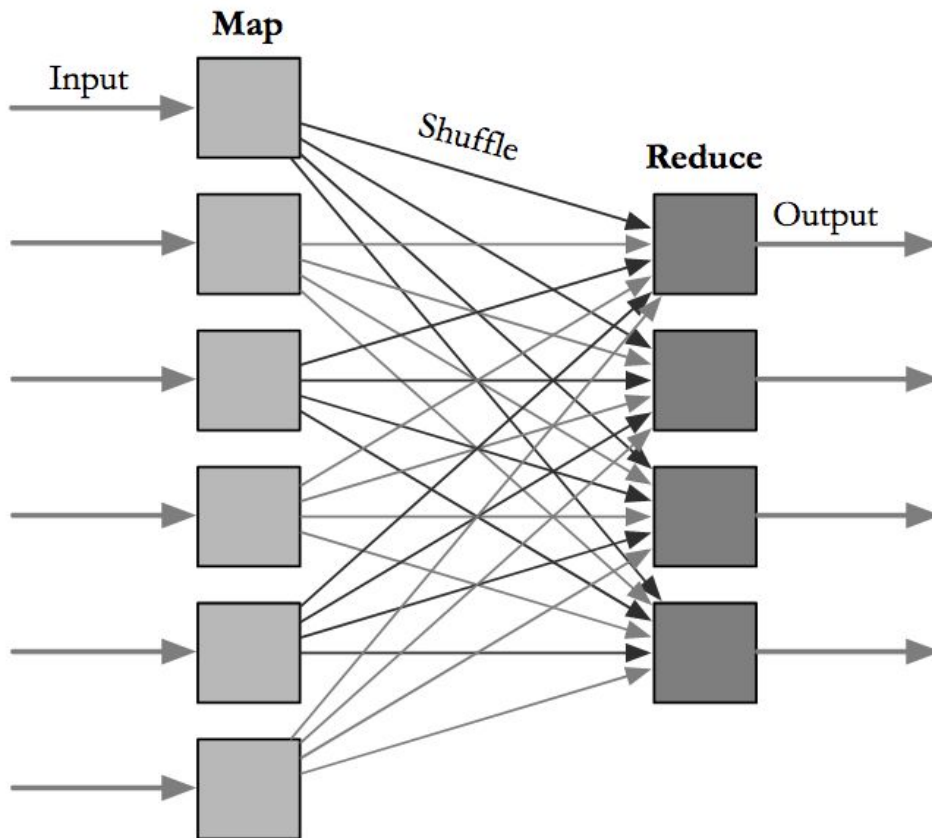
- Run `BuildIndex()` until memory runs out
- Write the **partial index** to disk (in lexicographic order) and start a new one in memory
- At the end, a number of partial indices exist on disk
- **Merge** pairs of partial indices until a single index remains



Distributed indexing



How can you employ Hadoop's map/reduce functionality to create an inverted index of e.g. CommonCrawl?



Index updates

Zipf's law

Collection term frequency decreases rapidly with rank

$$cf_i \propto \frac{1}{i}$$

Heap's law

The vocabulary size grows linearly with the size of the corpus

- **Static collections**: indexing as a one-off process
- Collections with few changes over time can be re-indexed every so often
 - Inverted file update not an option, as it requires writes in the middle of the file
- **Dynamic collections** change: Twitter and Ebay are extreme cases
 - Requires **multiple indices** (in memory/on disk) at the same time (plus a deleted doc. list) that are merged from time to time
 - Queries are scored against all indices and the deleted doc. list

Query processing



Memory vs. disk access?

Query processing

Document-at-a-time

Given a query, score a document, then move to the next document ...

Per document, all postings lists containing a query term are scanned to compute the $RSV(Q,D)$

Add the $RSV(Q,D)$ to priority queue

Term-at-a-time

Given a query, process one postings list (short to long) at a time

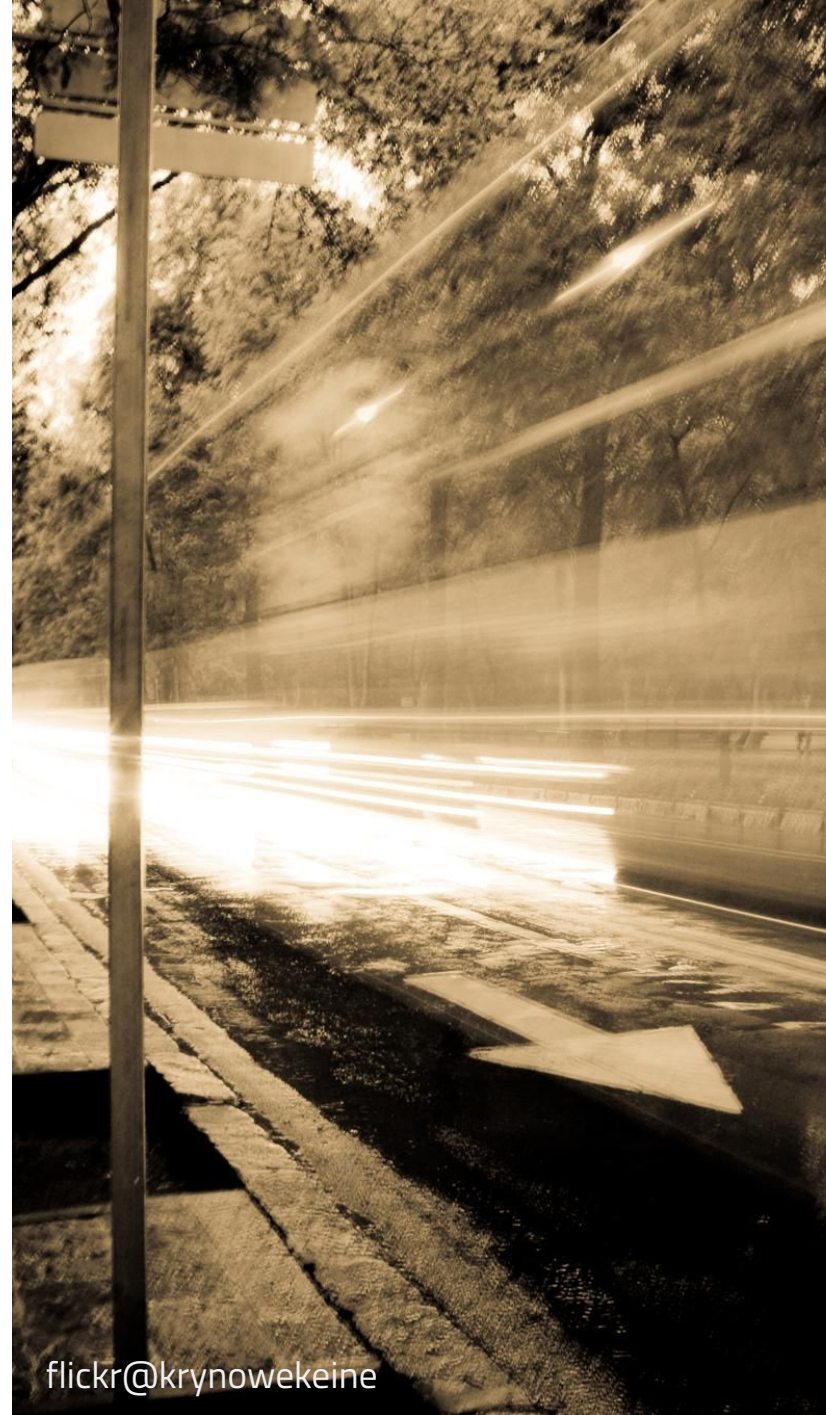
Store partial document scores in accumulators (one per document)

Compute final RSV values from accumulators and store in priority queue

More efficient query processing

Early stopping

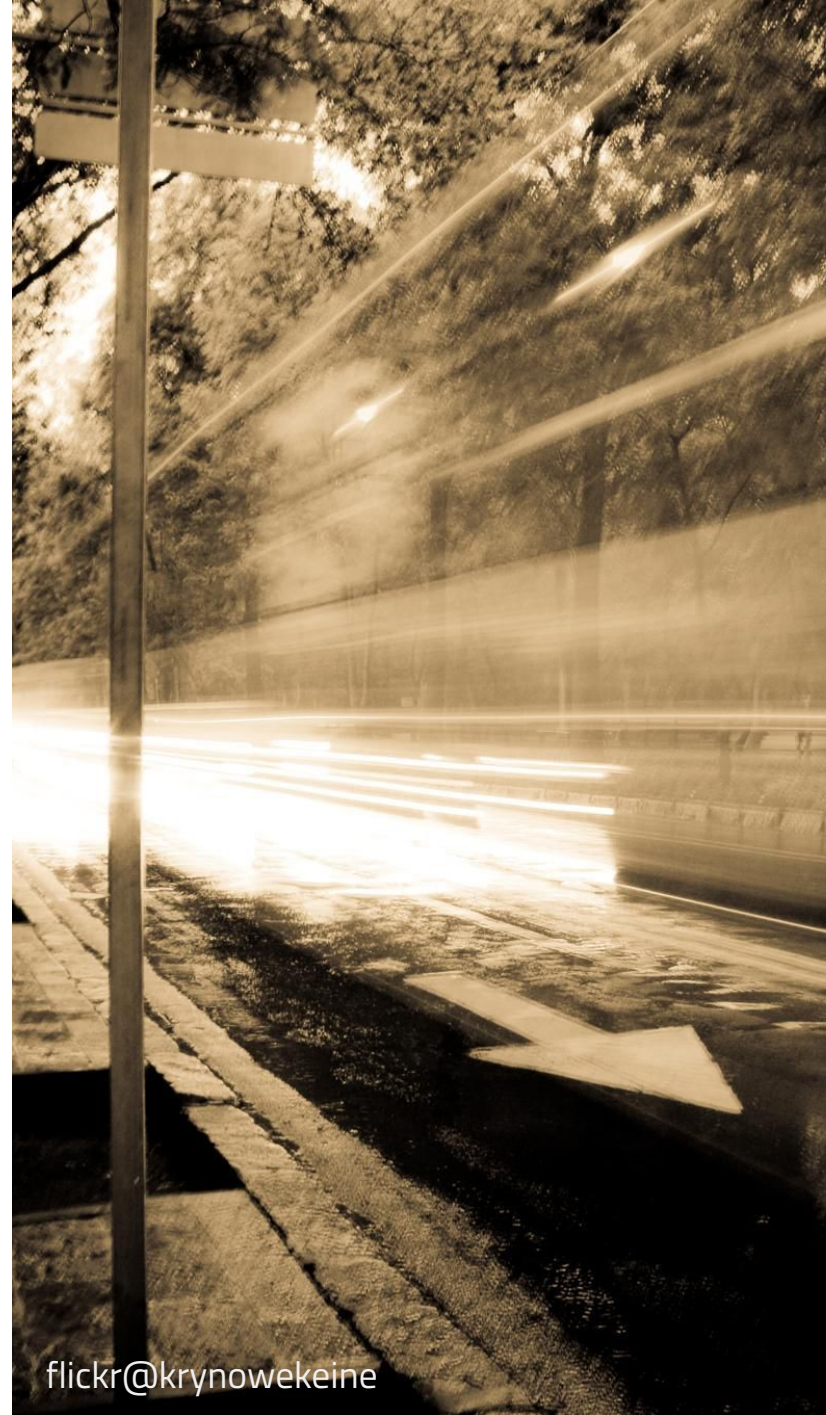
- Ignore some of the documents or terms
- Reduces impact of overly expensive queries, e.g. **the who** or **to be or not to be**
- Ideally in combination with postings list **impact ordering** (sort documents by their quality, update frequency, ...)
- Approximation



More efficient query processing

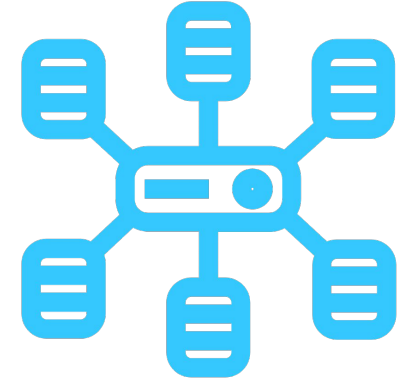
MAXSCORE

- Compute the largest partial score for documents with only some of the query terms
- If that score is lower than the k RSVs currently in the PriorityQueue ignore all documents that contain this subset of query terms
- Not an approximation



Distributed indexing

Overview



- We have already seen index *creation* across a cluster of machines
 - Several indexers must be coordinated for the final index
- Single-machine *query processing* is likewise not feasible for large corpora (e.g. CommonCrawl)
- Final index needs to be **partitioned**, it does not fit into a single machine
 - Splitting **documents** across servers
 - Splitting **index terms** across servers

Term-based index partitioning

- Known as **distributed global indexing**
- Query processing:
 - a. Queries arrive at the **broker** server which distributes the query and returns the results
 - b. Broker determines index server to collect all postings lists and compute the final document ranking
 - c. Results returned via the broker
- **Load balancing** depends on the distribution of query terms and its co-occurrences (query log analysis helps)

Document-based index partitioning

- Known as **distributed local indexing**
- Most common approach for distributed indexing today
- Query processing:
 - a. Every index server receives all query terms and performs a **local search**
 - b. Result documents are sent to the broker, which sorts them
- Issue: **maintenance of global collection statistics** inside each server (needed for document ranking)

Research in efficiency



What are we concerned with?

Metrics

Memory consumption vs. indexing time

Indexing throughput (n GB per hour/minute)

Efficiency vs. effectiveness: impact of pruning (#terms in pruned index) on retrieval effectiveness

Average time per query for “top-k retrieval”

Hardware software interplay

Is **compression** effective for current CPU architectures?

Effective **cache** population

Exploiting CPUs and **GPUs** to reduce query processing latency

Energy-efficient query processing (do not execute a query faster than required)

Deep learning

Neural rankers based on sparse representations for inverted indexing

(embeddings: every word matches every document to some extent)