IN4325

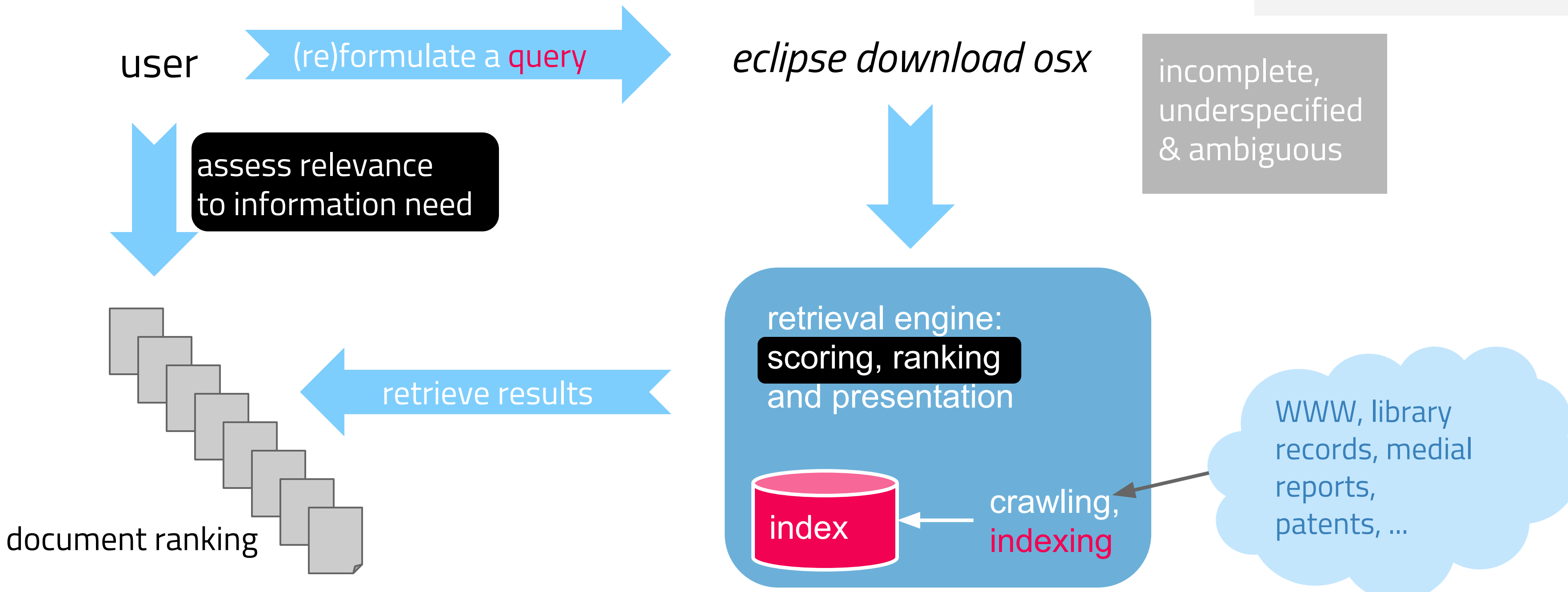
Indexing and query processing

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?*



today: indexing

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

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

Terminology

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

What's wrong with a file-based posting list?

termID (term identifier)

docID (document identifier) + other information = a posting

Term	termID		Postings
Albert	1	→	1 | 2 | 5 | 8
cell	2	→	2 | 3 | 7
cristo	3	→	2 | 6
dantes	4	→	1 | 3 | 5 | 7
edmond	5	→	4 | 5 | 7 | 9
imprisoned	6	→	1 | 7
prison	7	→	1 | 4 | 7 | 8

dictionary (entries sorted alphabetically)

postings lists (postings often ordered by docIDs)

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

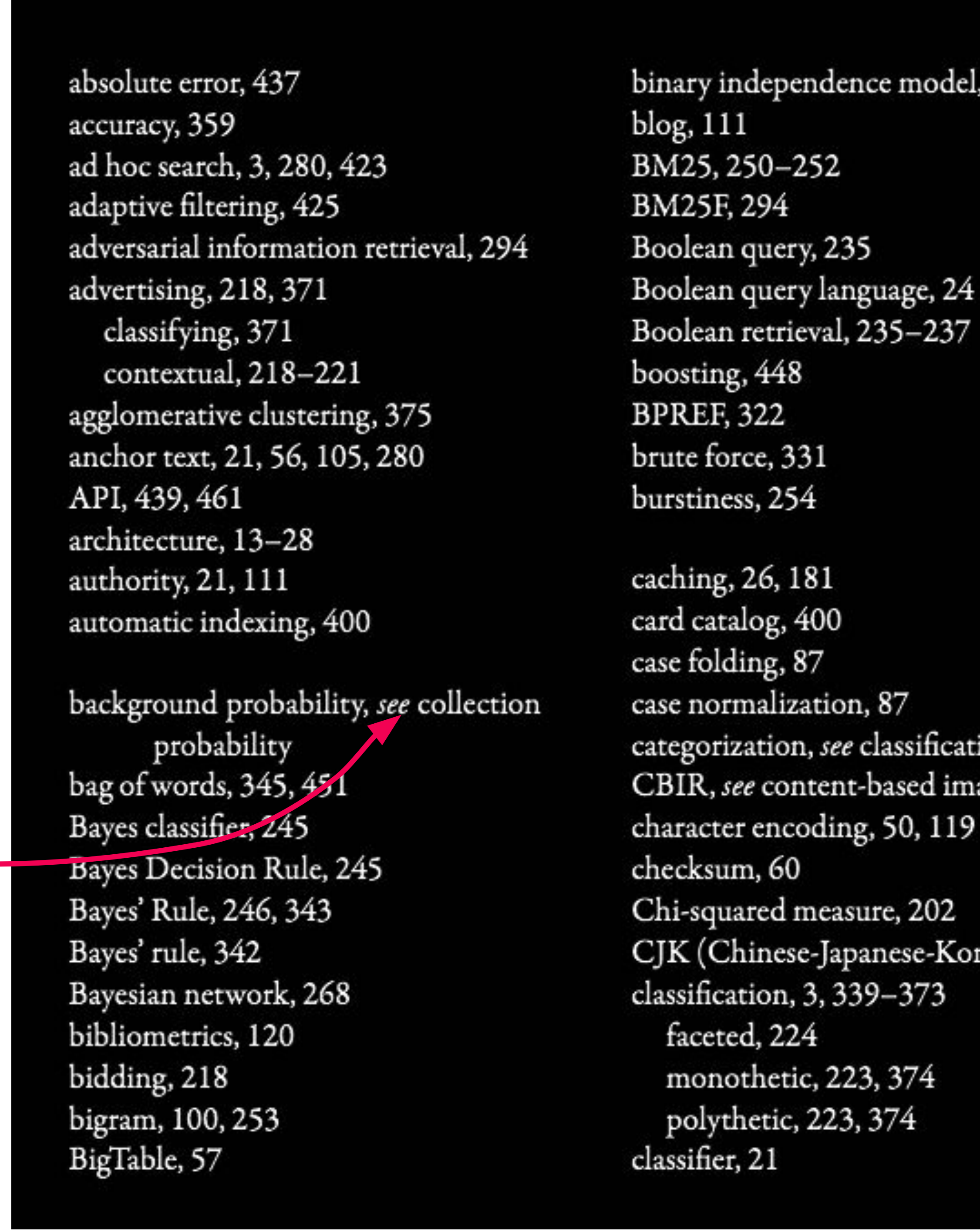
4) Index the documents

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



umbrella term for different data structures

Inverted index

Data structures depend on the retrieval models employed.

Lemur

Indexing as an offline process

↓

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

elastic

```
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"
        }
      }
    }
  }
}
```

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

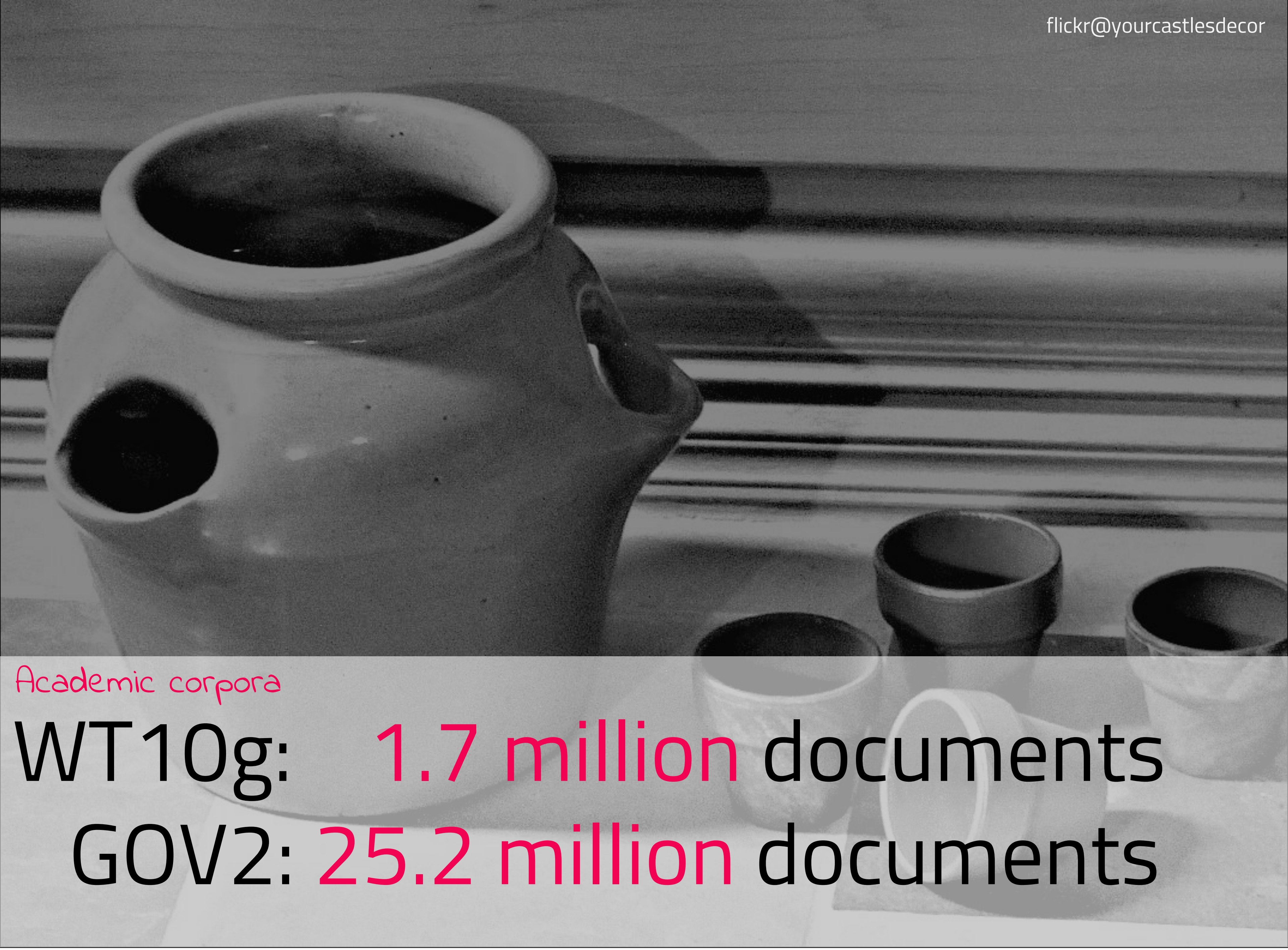
No queries

3.4 billion Web pages

270 TB uncompressed content

1.1 billion new URLs last month

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****.*

Chris Manning, Introduction to Information Retrieval, Chapter 2 (p. 35)

Hardware constraints to think about

- Disks maximize input/output throughput if contiguously stored data is accessed

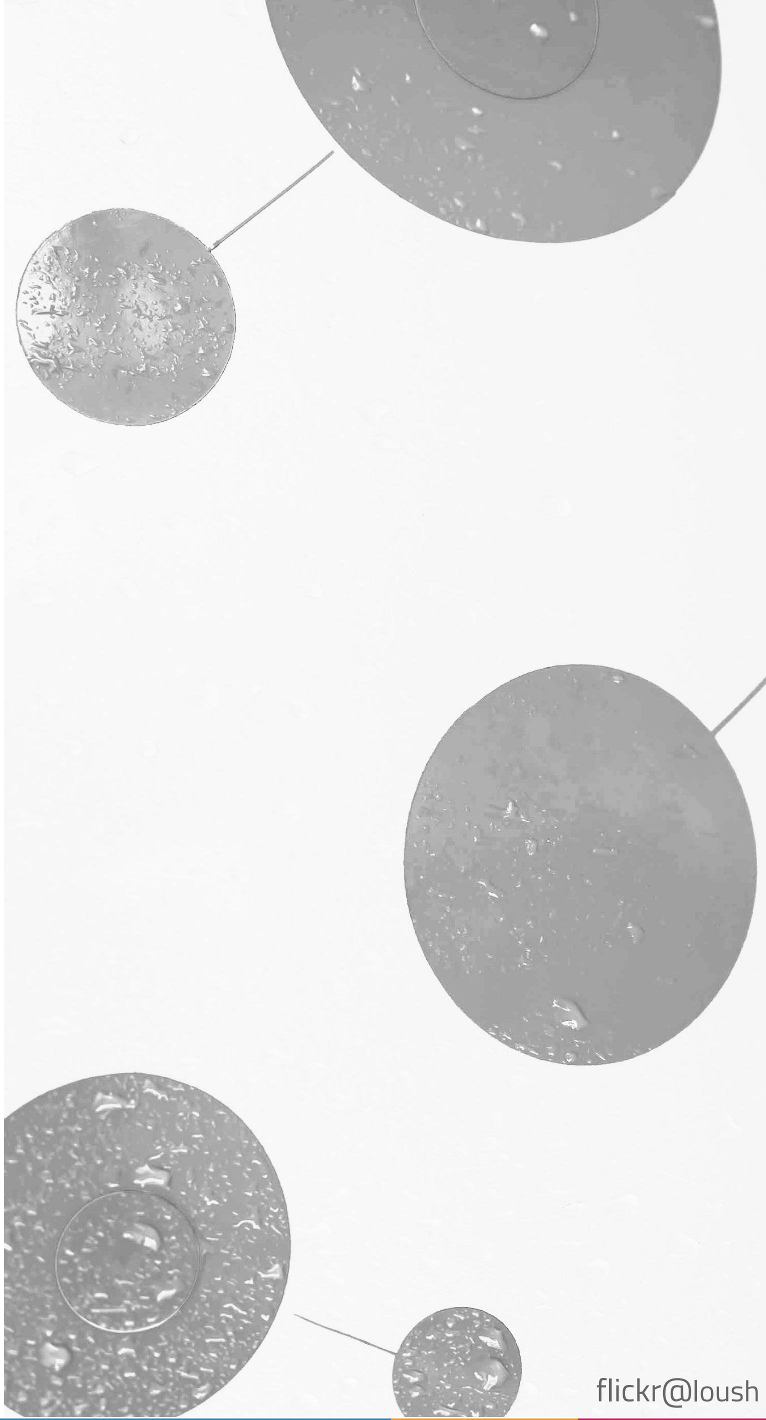
- Memory access is faster than disk access

- Operating systems read/write blocks of fixed size from/to disk

- Reading compressed data from disk and decompressing it is faster than reading uncompressed data from disk

Indexing in five steps

- Types of inverted indices
- Compression algorithms
- Index construction
- Query processing
- 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

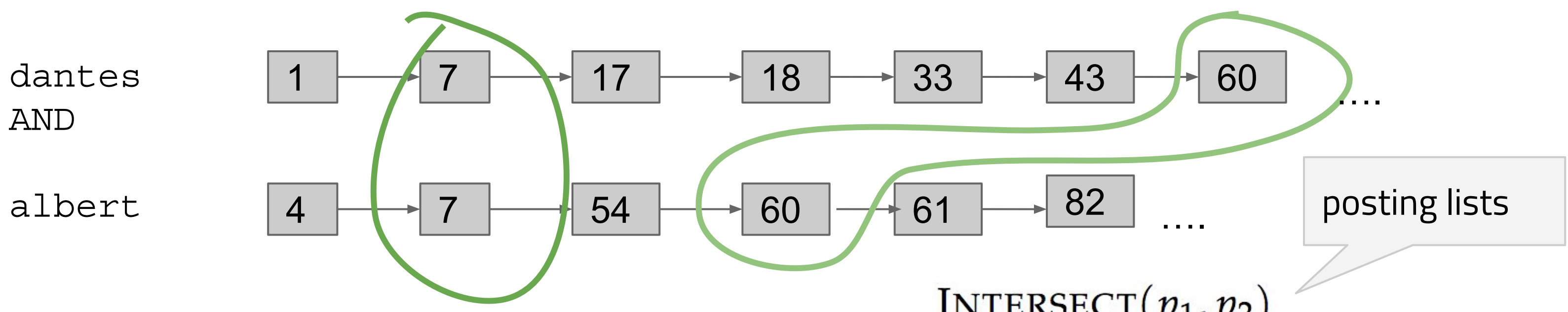
- Naive solution:

```
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 over posting lists

Dantes AND Albert




```
INTERSECT(p1, p2)
answer ← ⟨ ⟩
while p1 ≠ NIL and p2 ≠ NIL
do if docID(p1) = docID(p2)
      then ADD(answer, docID(p1))
           p1 ← next(p1)
           p2 ← next(p2)
      else if docID(p1) < docID(p2)
             then p1 ← next(p1)
             else p2 ← next(p2)
return answer
```

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

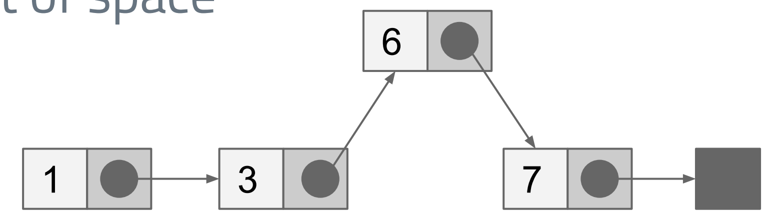
Posting lists data structures

Index needs to be optimized for:

- Storage and access efficiency

How to implement postings lists?

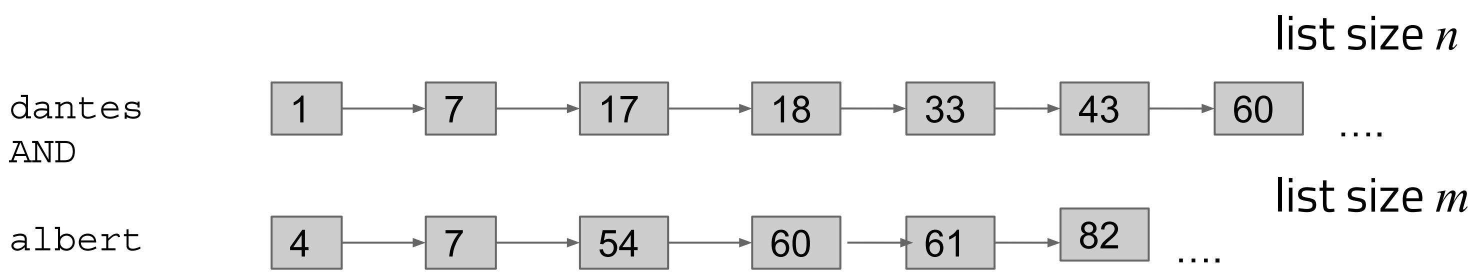
- Fixed length array: easy, wastes a lot of space
- Singly linked list: cheap insertion



- Variable length arrays
  - Require less space than linked lists (no pointers)
  - Allow faster access (contiguous memory increases)
  - Good if few updates are required

Boolean retrieval over posting lists

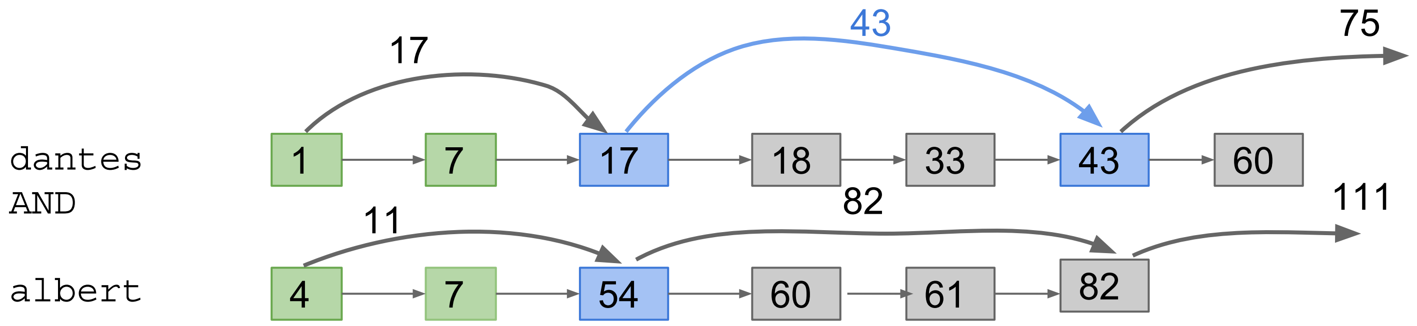
Skip pointers (created at indexing time)



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

Boolean retrieval over posting lists

Skip pointers are shortcuts



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

Are skip pointers useful for OR queries?
Is anything stopping us from conducting a binary search?

Boolean retrieval over posting lists

posting lists

common docID found in both lists

Increment posting list counter, skip if possible

```
IntersectWithSkips(p1, p2)
answer ← ⟨ ⟩
while p1 ≠ NIL and p2 ≠ NIL
do if docID(p1) = docID(p2)
      then Add(answer, docID(p1))
           p1 ← next(p1)
           p2 ← next(p2)
      else if docID(p1) < docID(p2)
              then if hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
                      then while hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
                              do p1 ← skip(p1)
                      else p1 ← next(p1)
              else if hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
                      then while hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
                              do p2 ← skip(p2)
                      else p2 ← next(p2)
return answer
```

Posting lists data structures

Skip pointers: where to place them

Tradeoff:

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

Heuristic: for posting lists of length L, use $sqrt(L)$ evenly spaced skip pointers (ignores particularities of the query term distribution)

Effective skip pointers are easy to create in static indices, harder when the posting lists are frequently updated

Positional postings

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 posed queries are explicit phrase queries)

Posting lists of the form $termID \rightarrow d1|d2|d3|\ldots$ do not provide sufficient granularity

- Require substantial **post-retrieval filtering**

Biword indices

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

Biwords: every 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

vocabulary

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); add a single term index as solution

Arbitrary phrases are usually not indexed, **vocabulary sizes** increase greatly

The Count of Monte Cristo
~50K lines of text

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

Positional indices

Most common index type

For each term, postings are stored with frequency values

to occurs 993,427 times in the corpus

$$
\begin{aligned}
&\textsf{to}, 993427: \\
&\quad \langle\, 1, 6: \langle 7, 18, 33, 72, 86, 231\rangle; \\
&\quad\;\; 2, 5: \langle 1, 17, 74, 222, 255\rangle; \\
&\quad\;\; 4, 5: \langle 8, 16, 190, 429, 433\rangle; \\
&\quad\;\; 5, 2: \langle 363, 367\rangle; \\
&\quad\;\; 7, 3: \langle 13, 23, 191\rangle; \ldots\rangle
\end{aligned}
$$

to occurs 6 times in document 1

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

$$
\begin{aligned}
&\textsf{be}, 178239: \\
&\quad \langle\, 1, 2: \langle 17, 25\rangle; \\
&\quad\;\; 4, 5: \langle 17, 191, 291, 430, 434\rangle; \\
&\quad\;\; 5, 3: \langle 14, 19, 101\rangle; \ldots\rangle
\end{aligned}
$$

Source: Introduction to Information Retrieval, Manning et al. (p. 38)

Positional indices

To process a phrase query: `“to be or not to be”`

- Access the postings list for each term
- When merging (intersecting) the result list, check if the positions of the terms match the phrase query
  - Calculate offset between terms
  - Start with the least frequent term

Increased index size: the index is 2-4x larger than a non-positional index

Why not more? Position integers tend to be small; they are limited by the document length

In practice: combine biword and positional indices. Which queries should be processed which index type?

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: **hashes** or **search trees**

Choice depends on:

- Number of terms (keys)
- Frequency and type of changes (key insert/delete) in the index
- Frequency of key accesses

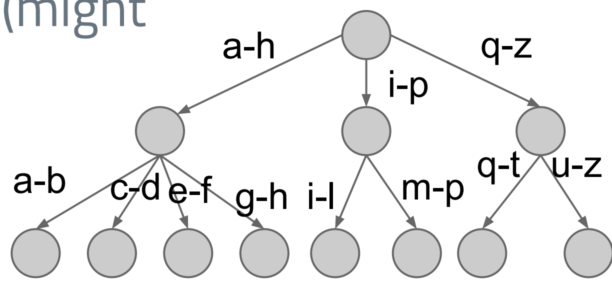
Dictionary lookup

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

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

Binary search trees overcome those issues. 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 sought (restoration vs restore)

Trailing wildcard query: `restor*`

single wildcard

- Search trees are perfect for this situation: walk along the edges and enumerate the W terms with prefix `restor`; followed by $|W|$ lookups of the respective posting lists to retrieve all docIDs

Wildcard queries

single wildcard

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

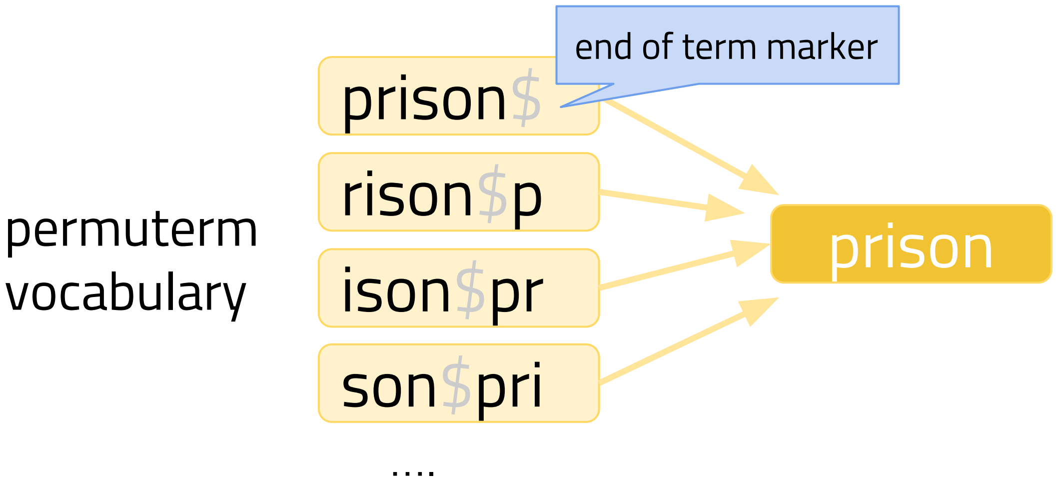
single wildcard

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

Multiple wildcards: Permuterm index

Dictionary increases substantially in size!



Query `pr*son → pr*son$`

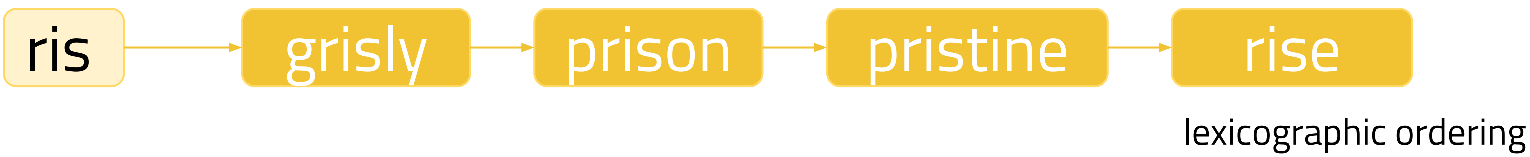
- Move * to the end: `son$pr*`
- Look up the term in the permuterm index (search tree)
- Look up the found terms in the standard inverted index

Query `pr*s*n → pr*s*n$`

- Start with `n$pr*`
- Filter out all results not containing ‘s’ in the middle
- Look up the found terms in the standard inverted index

Multiple wildcards: N-gram index

Each N-gram in the dictionary points to all terms containing the N-gram



Wildcard query: `pr*on`

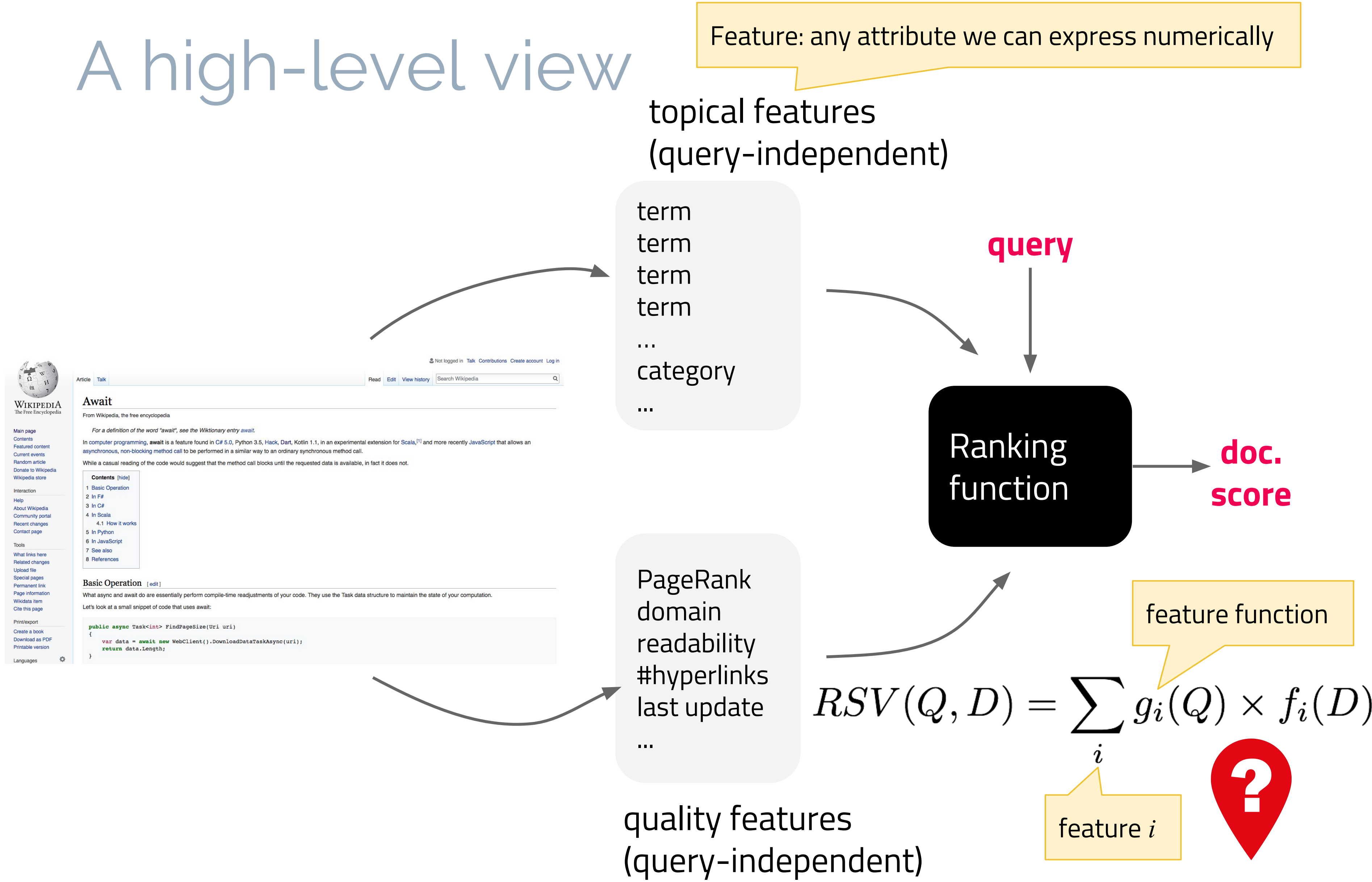
- Boolean query `$pr AND on$`
- Look up in a 3-gram index yields a list of matching terms
- Look up the matching terms in a standard inverted index

Wildcard query: `red*`

- Boolean query `$re AND red` (also retrieves retired)
- Post-filtering step to ensure enumerated terms match

Beyond boolean retrieval

A high-level view

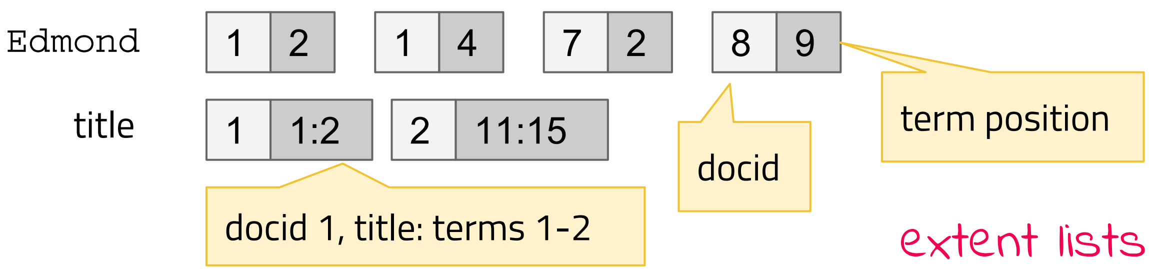


$$RSV(Q, D) = \sum_i g_i(Q) \times f_i(D)$$

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, main, 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

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:

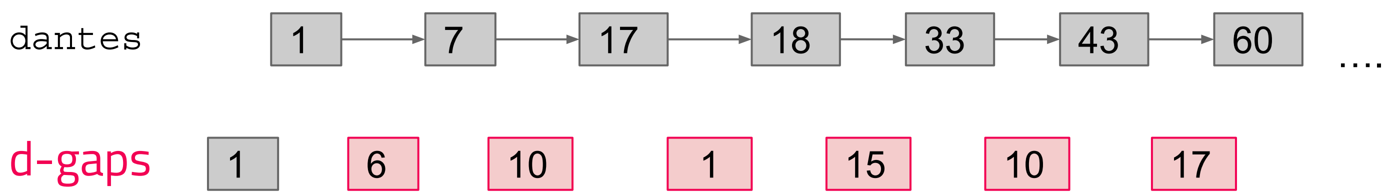
e.g. word counts (docids) in postings lists tend (not) to be small.

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



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)
```

Have we gained anything? We still have a list of integers - however, those integers are mostly **small** (lets compress those!)

Unary code

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

Number	*Symbol*
0	0
1	10
2	110
5	111110

why can't we just use binary code?

unambiguous decoding is not possible

10111010111010110100

Unary encoding is efficient for 0/1 but not 1023 (requires 10 bits in binary vs. 1024 in unary code)

However: it is **unambiguous**, convenient and easy to decode.

Elias-γ code

Idea: combine the strength of unary and binary code

To encode a number k we compute:

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

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

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

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

Unary code

Binary code

= Elias-γ code

k	*kd*	*kr*	*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

Elias-γ code

Idea: combine the strength of unary and binary code

To encode a number k we compute:

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

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

Unary code

Binary code

= Elias-γ code

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

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

Refinement: Elias-δ code

Elias-γ is not ideal for inputs that *may* contain large numbers

A single change: 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**.

Source: Search Engines - IR in Practice, Croft et al. (p. 147)

Source: Search Engines - IR in Practice, Croft et al. (p. 149)

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.])
(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

brackets only for readability

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

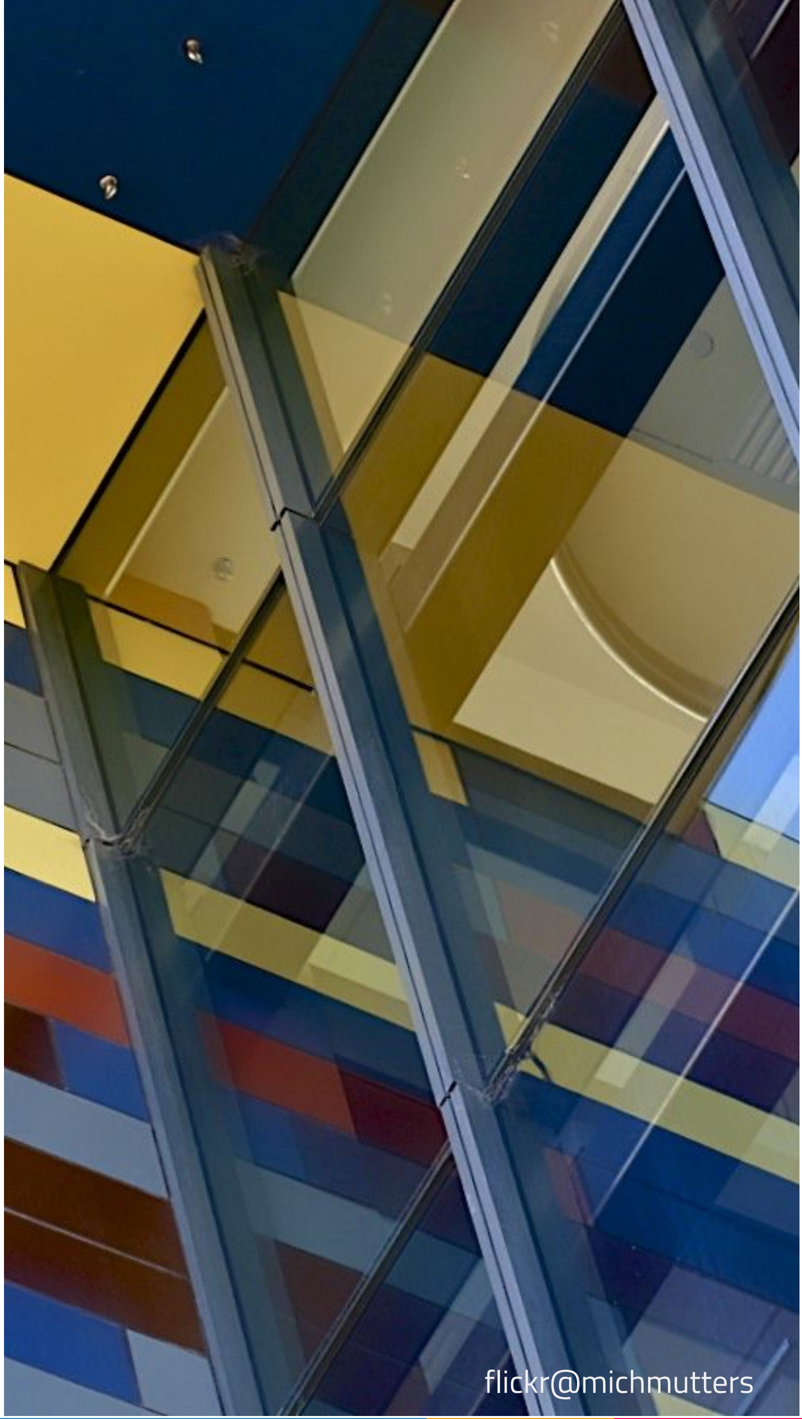
How can we compute the inverted file
when our document corpus has
Terabytes or Petabytes of text?

Increasing complexity

In-memory index construction

Single machine (disk-based) index construction

Cluster-based index construction (corpus does not fit onto a single machine)


flickr@michmutters

In-memory indexing

```
procedure BUILDINDEX(D)                          ▷ D is a set of text documents
    I ← HashTable()                              ▷ Inverted list storage
    n ← 0                                        ▷ Document numbering
    for all documents d ∈ D do
        n ← n + 1
        T ← Parse(d)                             ▷ Parse document into tokens
        Remove duplicates from T
        for all tokens t ∈ T do
            if I_t ∉ I then
                I_t ← Array()
            end if
            I_t.append(n)
        end for
    end for
    return I
end procedure
```

All posting lists are maintained in memory

Requires additional effort to parallelize

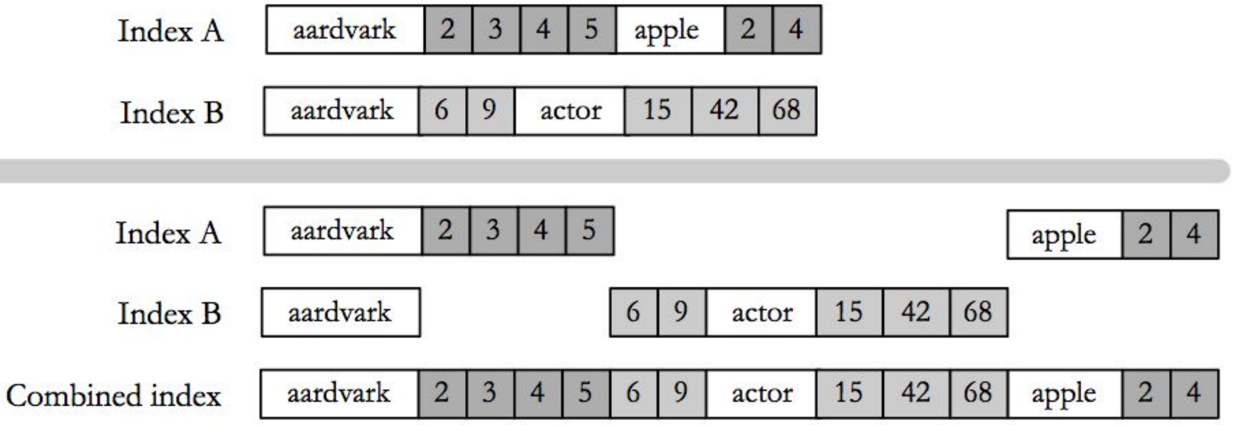
Source: Search Engines - IR in Practice, Croft et al. (p. 157)

Source: Search Engines - IR in Practice, Croft et al. (p. 158)

Using the disk …

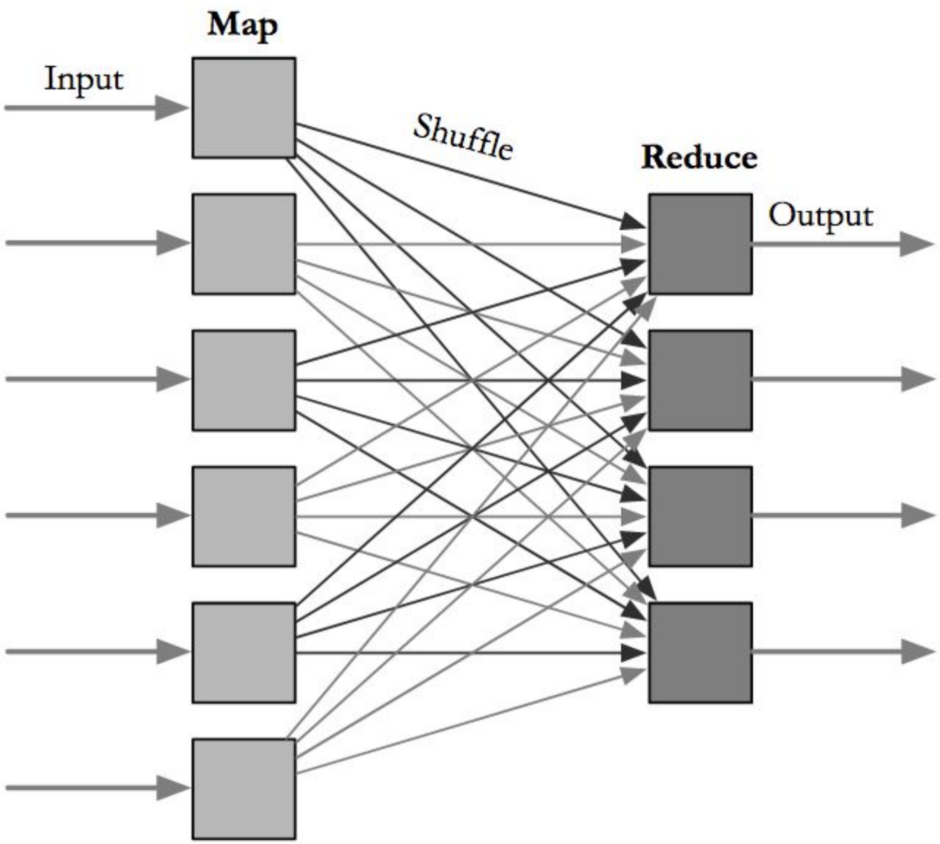
- Run `BuildIndex()` until memory runs out
- Write the **partial index** to disk (in lexicographic order) and start a new one in memory
- At the end, a number of partial indices exist on disk

- **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?



flickr@techmsg

Index updates

- **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

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

KBacon @Dontm8kamricaH8 · 17s

Hope Hicks is out — here are all the casualties of the **Trump** administration so far a.msn.com/01/en-us/AAozb… #WorstPresidentEver

Hope Hicks is out — here are all the casualties of the Trump administ…

Skye Gould/Business Insider The White House announced on Wednesday that communications director Hope Hicks

msn.com

Nightengalejml2 @54nightengale · 17s

Jared Kushner is an easy mark on the world stage.

Other countries have good reasons to think that **Trump**'s son-in-law and senior

Impressive, considering the 500+ million tweets a day!

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 posting 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 posting 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 (DAAT) or terms (TAAT)
- 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 inversion

- 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:
  - Queries arrive at the **broker** server which distributes the query and returns the results
  - Broker determines index server to collect all postings lists and compute the final document ranking
  - Results returned via the broker

- **Load balancing** depends on the distribution of query terms and its co-occurrences (query log analysis can help here)

Document-based index partitioning

- Known as "**distributed local indexing**"
- Most common approach for distributed indexing today
- Query processing:
  - Every index server receives all query terms and performs a **local search**
  - 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)

Predict and approximate

Selective query rewriting based on efficiency predictions

Simulation and **cost models**