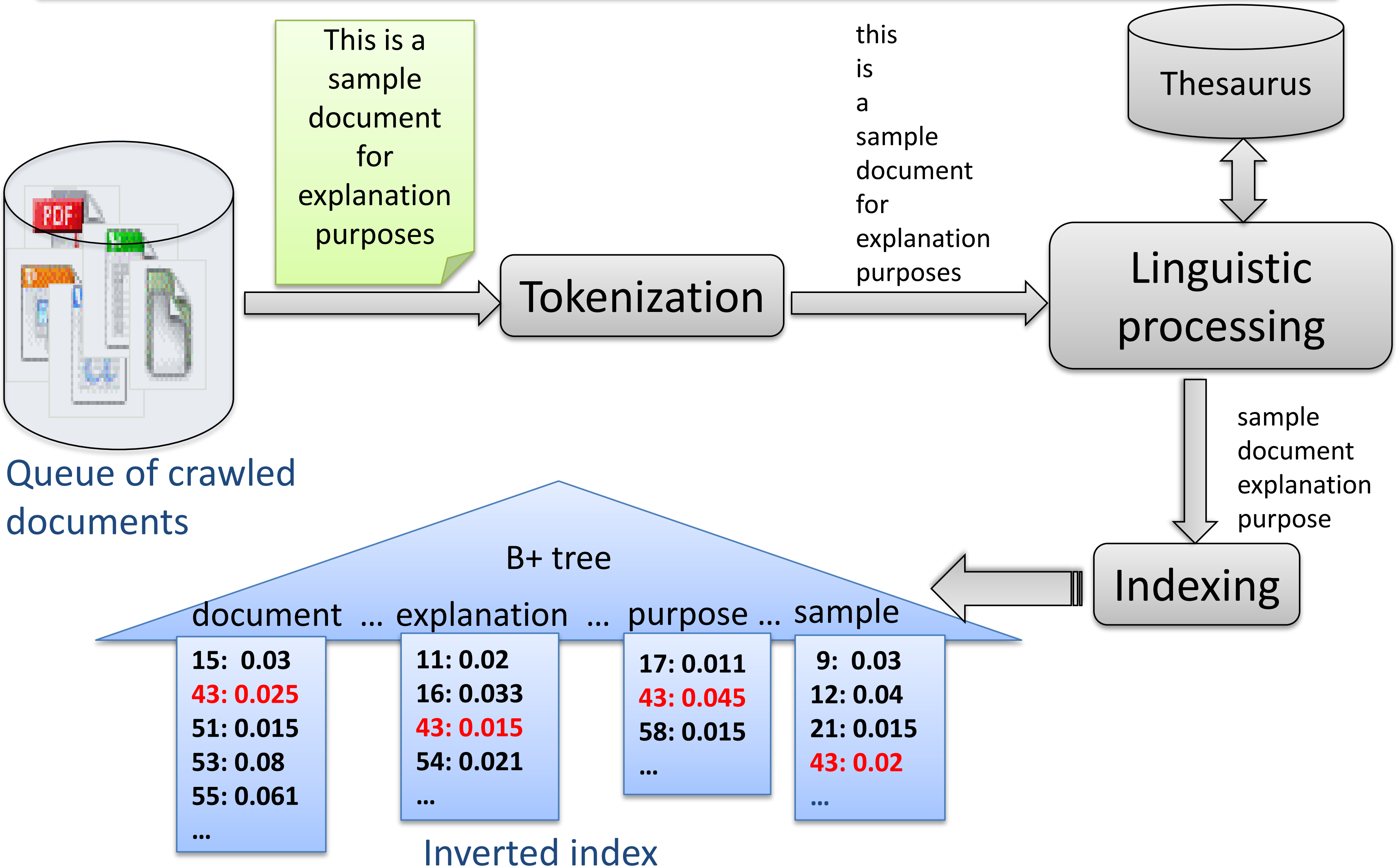
INVERTED INDEX CONSTRUCTION

Outline

- Intro
- Basics of probability and information theory
- Retrieval models
- Retrieval evaluation
- Link analysis
- **From queries to top-k results**
  - Query processing
  - **Index construction**
  - Top-k search
- Social search

Overview

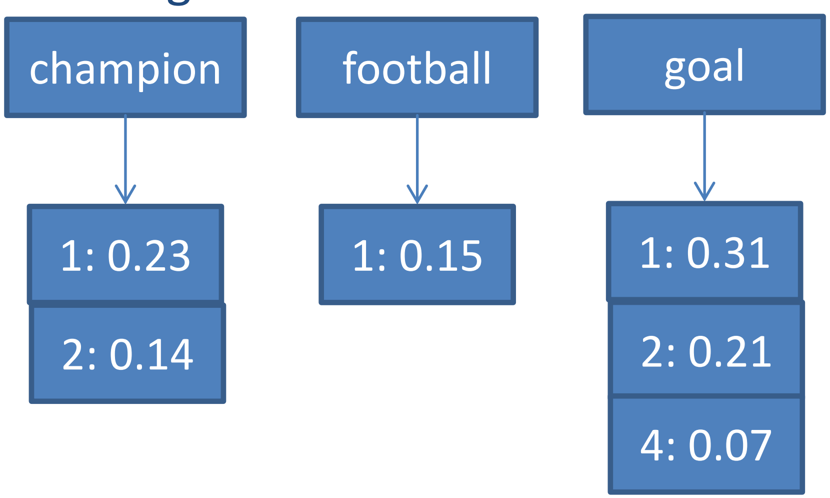

This is a sample document for explanation purposes
Tokenization
this
is
a
sample
document
for
explanation
purposes
Thesaurus
Linguistic processing
sample
document
explanation
purpose
Indexing
Queue of crawled documents
B+ tree
document … explanation … purpose … sample
15: 0.03
43: 0.025
51: 0.015
53: 0.08
55: 0.061
…
11: 0.02
16: 0.033
43: 0.015
54: 0.021
…
17: 0.011
43: 0.045
58: 0.015
…
9: 0.03
12: 0.04
21: 0.015
43: 0.02
…
Inverted index

Inverted index from the term-document matrix

- How to store a **realistic term-document matrix** with **millions of terms** and **hundreds of millions of documents**?

- Obviously a document contains relatively few terms.
  → Document vectors contain many zeros.
    → The whole matrix contains **a lot more** zeros than ones.

- Store for each term only the IDs of the documents in which it occurs, along with scores.



	d_1	d_2	d_3	d_4	d_5	d_6
champion	3	2	0	0	0	0
football	2	0	0	0	0	0
goal	4	3	0	1	0	0
law	0	0	2	3	0	0
party	0	0	6	5	0	0
politician	0	0	4	4	0	0
rain	0	0	0	0	3	3
score	4	5	0	0	0	0
soccer	0	3	0	0	0	0
weather	0	0	0	0	5	4
wind	0	1	0	0	2	3

Important steps for index constructions

- Sort documents by terms.
- Merge multiple term occurrences in a single document but maintain position information and add frequency information.
- Construct corpus vocabulary with entries of the form $(term, \#docs, corpus\_count)$
- Construct for every term postings with entries of the form $(docID, count, list[\mathrm{pos1}, \mathrm{offsets}..])$

  Why are position-based postings better than postings that store biwords or longer phrases (e.g., ‘stanford university’ or ‘hasso plattner institute’)?

- All steps involve distributed computations (e.g., through MapReduce methods)

Example

	d_1	d_2	d_3	d_4	d_5	d_6
champion	3	2	0	0	0	0
football	2	0	0	0	0	0
goal	4	3	0	1	0	0
law	0	0	2	3	0	0
party	0	0	6	5	0	0
politician	0	0	4	4	0	0
rain	0	0	0	0	3	3
score	4	5	0	0	0	0
soccer	0	3	0	0	0	0
weather	0	0	0	0	5	4
wind	0	1	0	0	2	3

term	**#docs**	**#**
champion	2	5
football	1	2
goal	3	8
law	2	5
party	2	11
politician	2	8
rain	2	6
score	2	9
soccer	1	3
weather	2	9
wind	3	6

Vocabulary

docID	**freq**
1	3
2	2
1	2
1	4
2	3
4	1
3	2
4	3
3	6
4	5
. . .	. . .

Pointers

Frequency-based postings (offsets omitted)

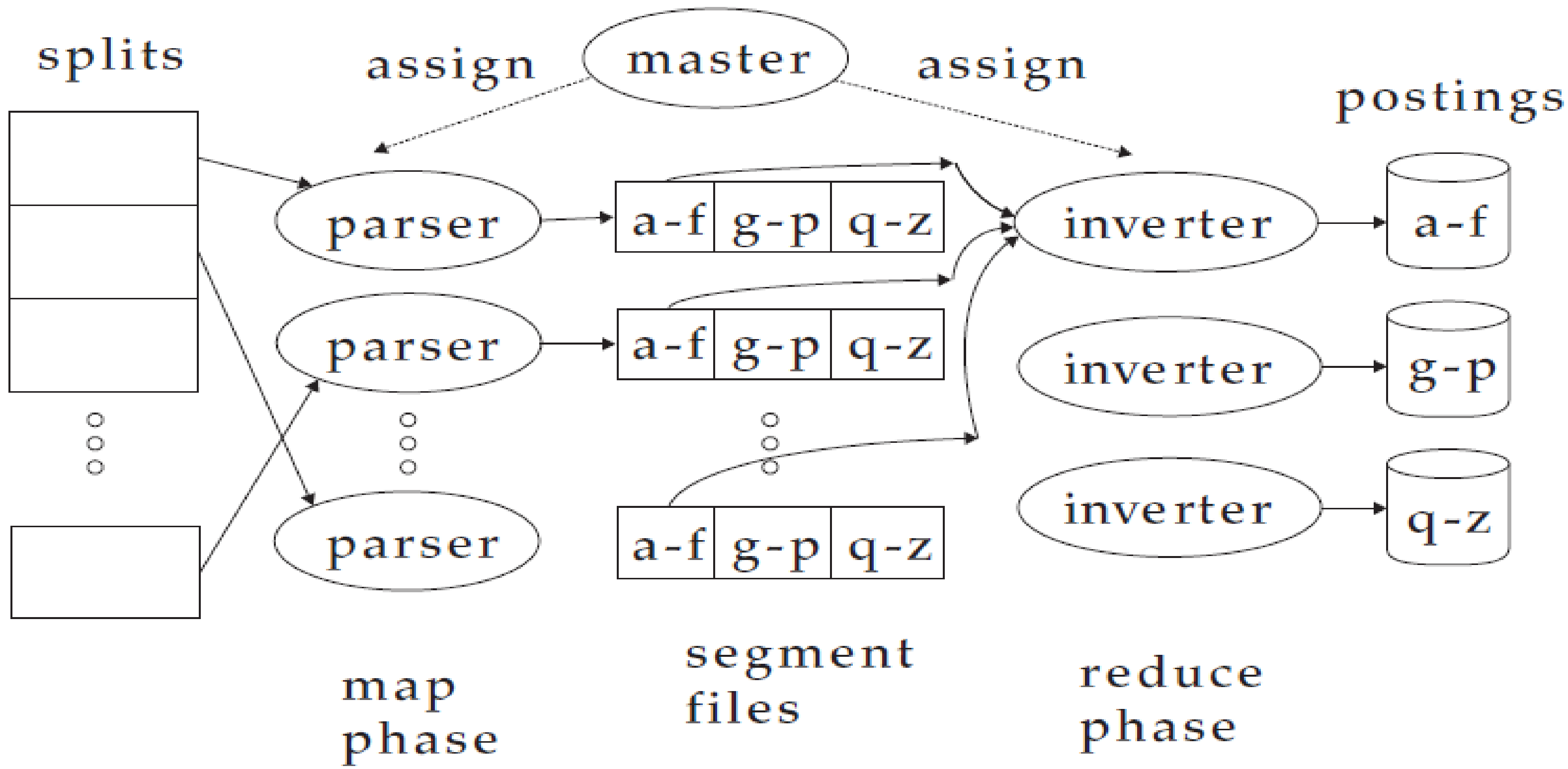
Distributed index construction with MapReduce

- Programming paradigm for scalable, highly parallel data analytics
- Scheduling, load balancing and fault tolerance are core ingredients
- Enables distributed computations on 1000's of machines
- Programming based on key-value pairs:

$$Map\colon\ K \times V \ \rightarrow (L \ \times W)^*$$
$$(k, v) \longmapsto (l_1, w_1), (l_2, w_2), \ldots$$

$$Reduce\colon\ L \ \times W^* \ \rightarrow W^*$$
$$l, (x_1, x_2, \ldots) \ \longmapsto y_1, y_2, \ldots$$

Possible MapReduce Infrastructure for Indexing



Source: [Introduction to Information Retrieval]

- MapReduce implementations: [PIG (Yahoo)], [Hadoop (Apache)], [DryadLinq (Microsoft)], [Facebook Corona]

TF computation with MapReduce

- Step 1

  Map:

  (docID, content) → {((term, docID), 1), …}

  Reduce:

  ((term, docID), {1,…}) → {((term, docID), count)}

- Step 2

  Map:

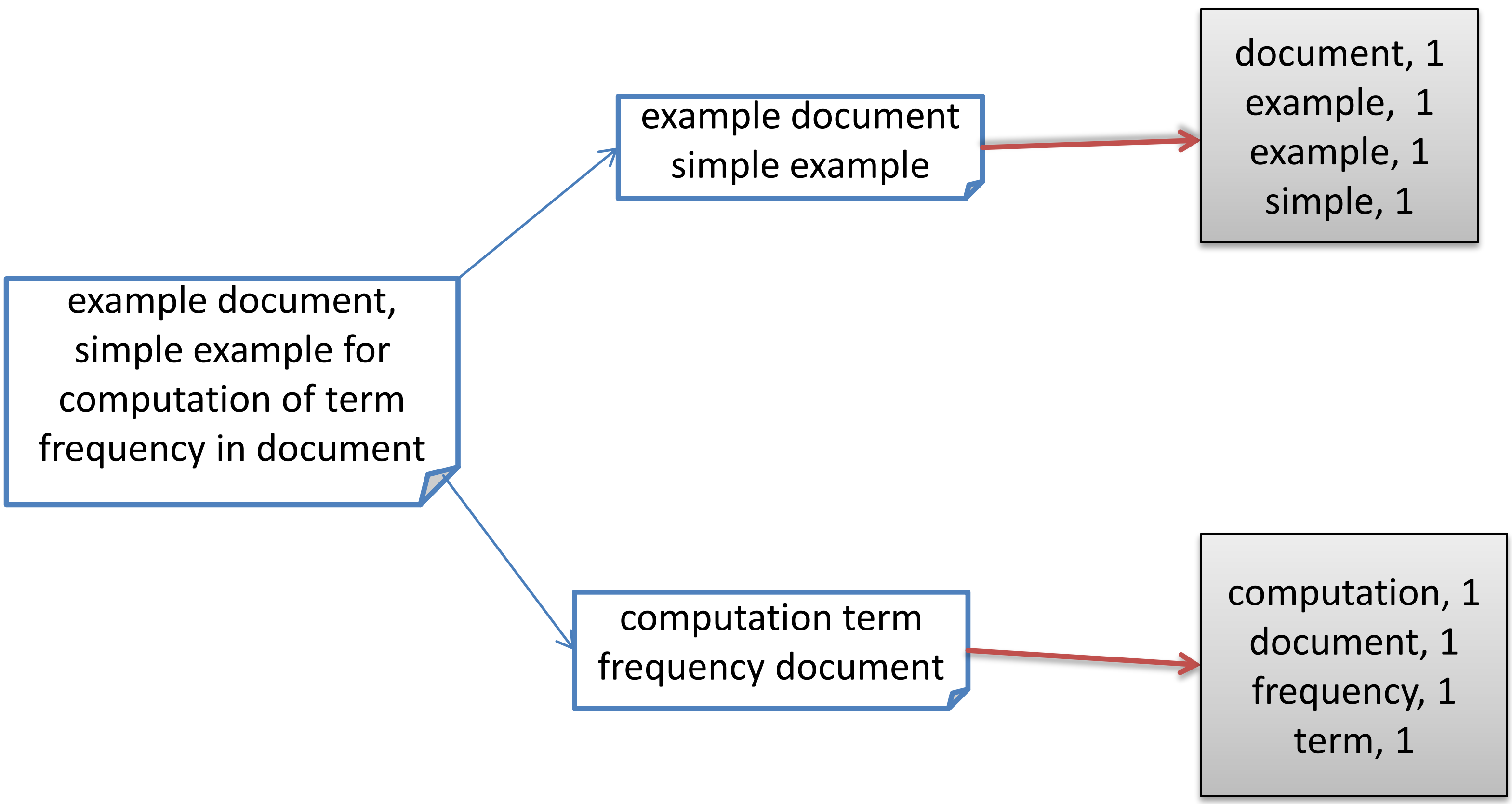
  ((term, docID), count) → {(docID, (term, count))}

  Reduce:

  (docID, {(term, count), …}) → {((docID, term), (count/doc_length)),…}

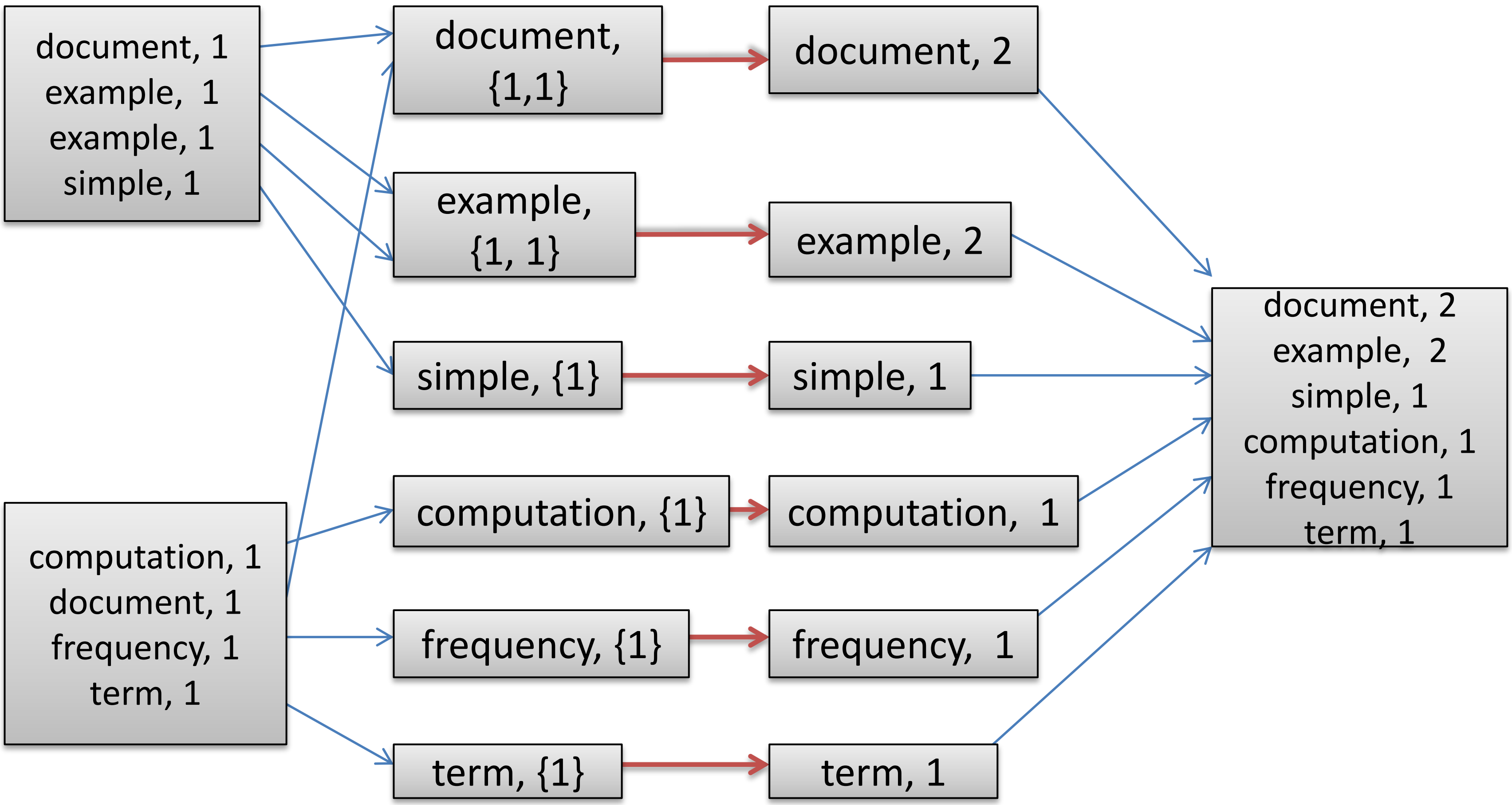
Example (1)

- Computing term counts: Map



Example (2)

➢ Computing term counts: Reduce



Size estimation for the data to be indexed

- 30 billion documents
- On avg. term occurs in ~ 100 documents
- 10 Mio. distinct terms

  → ~ 3×10^{12} entries for the postings

  → 10 Mio. entries for the vocabulary

- Assume ~5 Bytes per entry

  → ~ 15 TB in total

Question:

- How are the vocabulary and the postings stored?

term	#docs	#
champion	2	5
football	1	2
goal	3	8
law	2	5
party	2	11
politician	2	8
rain	2	6
score	2	9
soccer	1	3
weather	2	9
wind	3	6

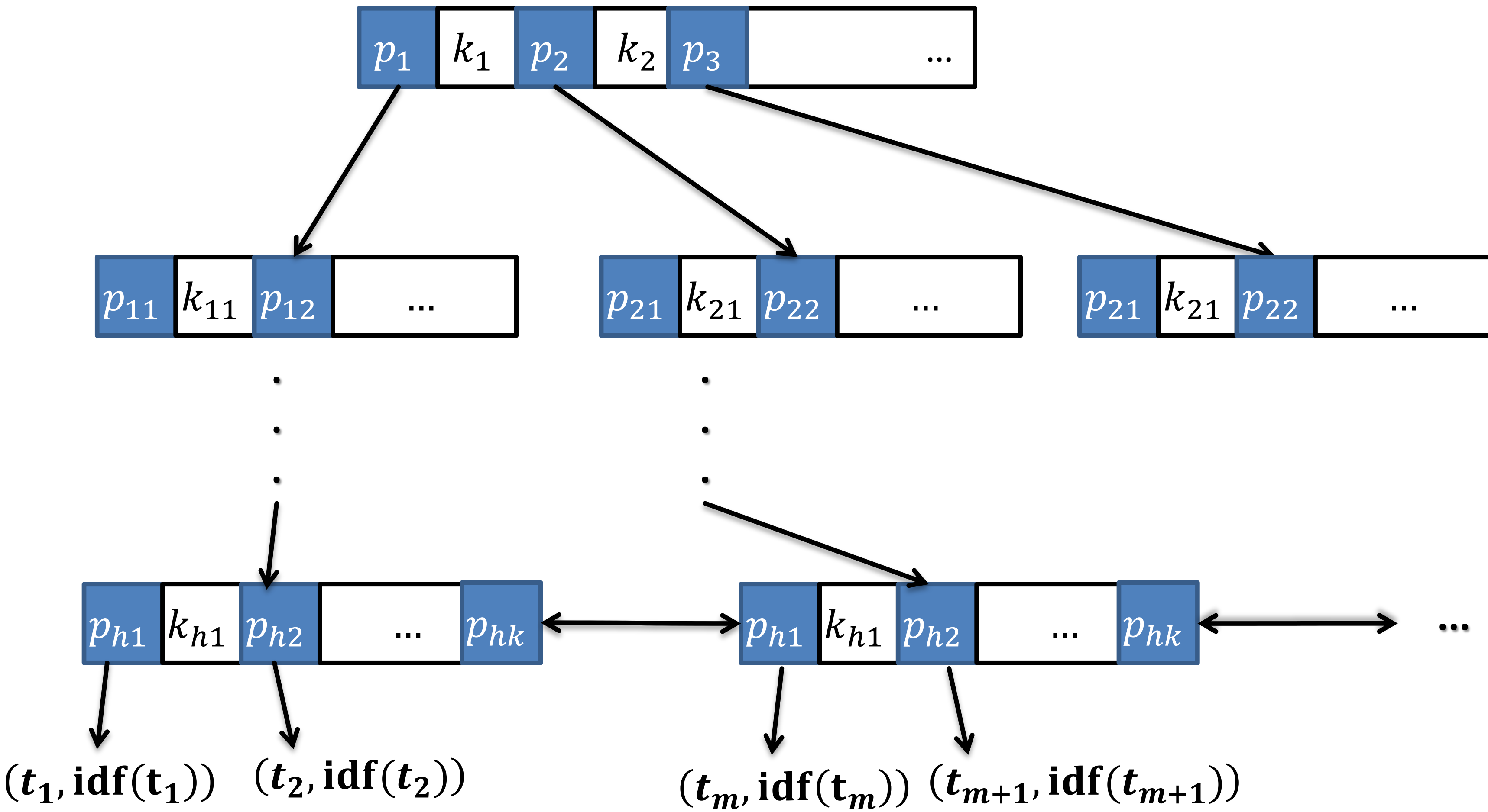
Vocabulary

docID	freq
1	3
2	2
1	2
1	4
2	3
4	1
3	2
4	3
3	6
4	5
. . .	. . .

Frequency-based postings (offsets omitted)

Storing the vocabulary: B+ trees

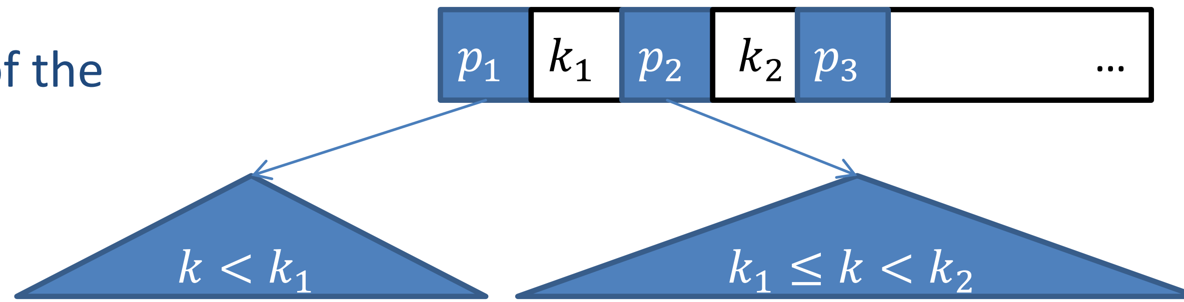
- Balanced search tree over the key space with high node fanout



Properties of B+ trees

- Every B+ tree is balanced

- Ordered partitioning of the key space



- In a B+ tree of order n (i.e., with fanout size n) every internal node, exept the root, has m children, with $\lceil n/2 \rceil \leq m \leq n$

- For the root: $2 \leq m \leq n$

- For the leaf nodes: $\lfloor n/2 \rfloor \leq m \leq n-1$

→ How could the insertion, deletion of keys be done?

Properties of B+ trees

- The maximum number of entries stored in B+ tree of order n and height h is $n^h - n^{h-1}$

  → a 4-level B+ tree of order n =100 would be sufficient to store 10 Mio. term keys

- The minimum number of entries stored in B+ tree of order n and height h is $2\left\lceil\frac{n}{2}\right\rceil^{h-1}$

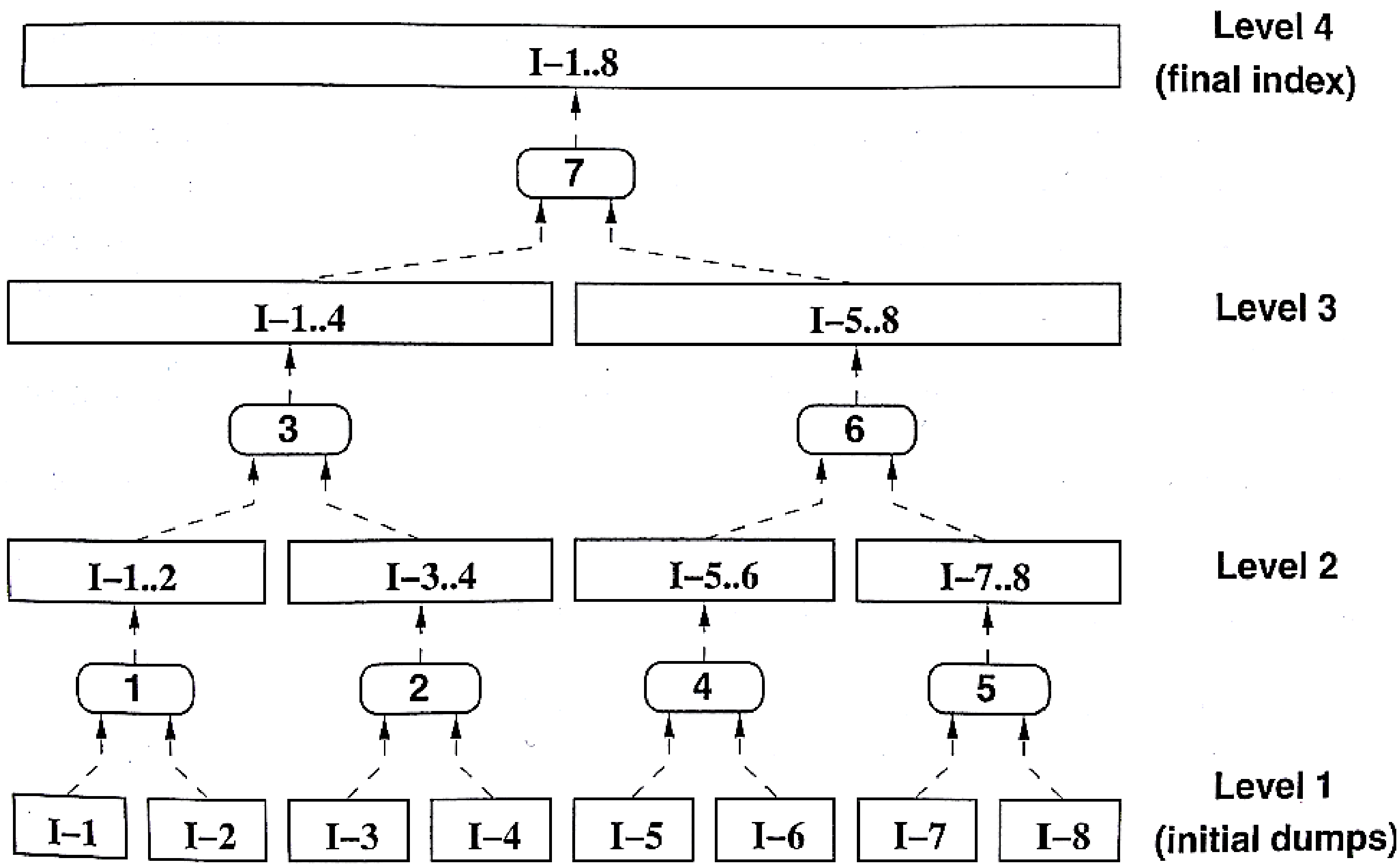
- Space required: $O(|K|)$, where K is the set of keys
- Insertion, deletion, finding: $O(\log_n(|K|))$

- Typically, the upper levels (up to the leaf level) of the B+ tree are loaded in main memory, the information linked with the leaves resides on disk.

B+ tree construction through bulk-loading

- Sort the entries by key values.
- Start with empty page as root node and insert a pointer to the first page of entries.
- Continue with the next page, insert its smallest key value into the root as separation key and insert pointer to this page. Repeat this step until the root is full.
- When the root is full, split it and create a new root.
- Keep inserting entries into the right most index node above the leaves, split the node when it is full and continue recursively

Index merging



Source: [Modern Information Retrieval]()

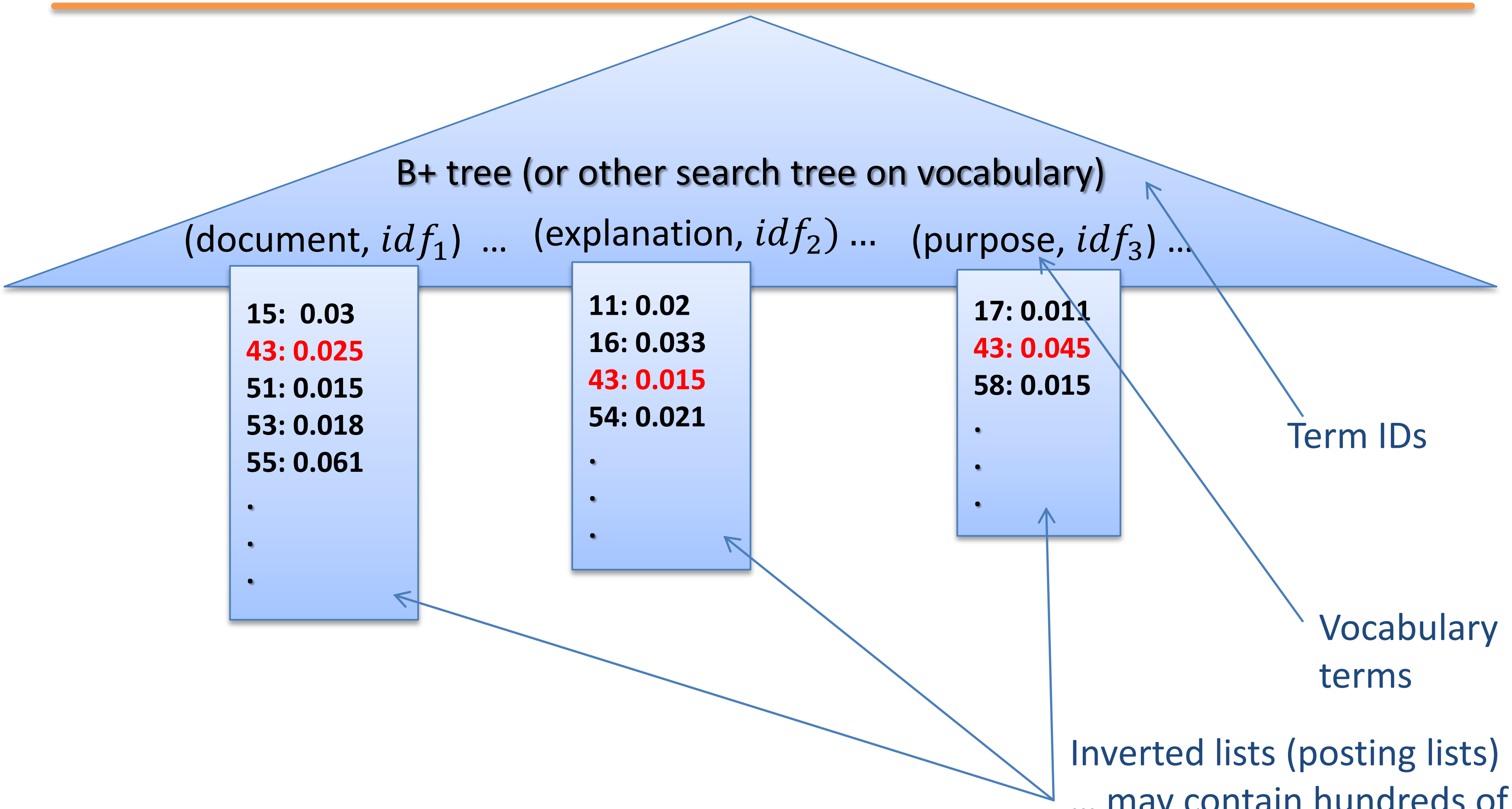
Dynamic Index

- On the web, pages are constantly added, deleted, modified

- Solution
  - Use index I_0 for the static pages
  - Use index I_+ for documents that are added
  - Use index $I_\sim$ for documents that are frequently modified
  - Use index I_- for documents that are deleted

  - Complete index: $(I_0 \cup I_+ \cup I_\sim) \backslash I_-$

Final Index



- How to store the vocabulary efficiently?

Vocabulary compression (1)

- With naive dictionary storage:

term	document frequency	pointer to postings list
a	656,265	⟶
aachen	65	⟶
...	...	...
zulu	221	⟶
20 bytes	4 bytes	4 bytes

Source: Introduction to Information Retrieval

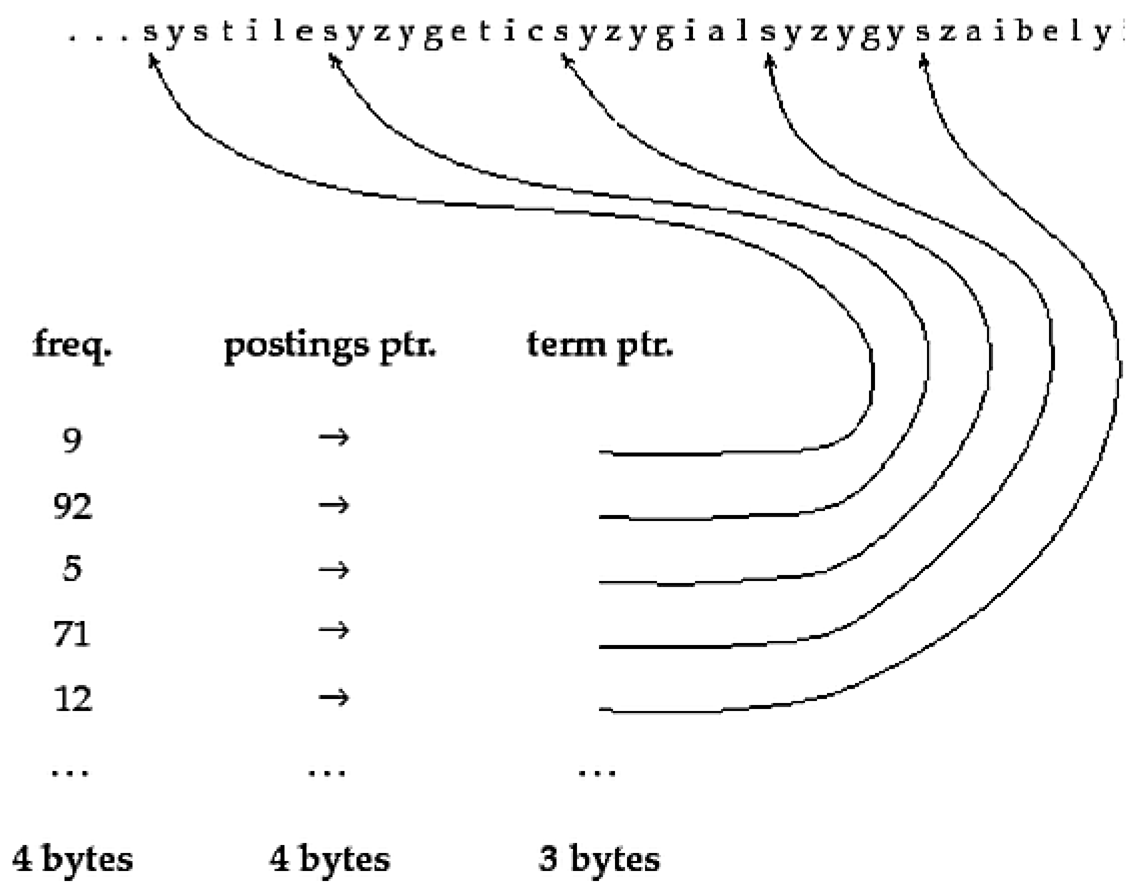
- In Unicode: (2×20 + 4 + 4) bytes per term
- For 10 Mio. terms: ~ 460 MB needed

  → fixed-width entries too wasteful

Vocabulary compression (2)

- Better strategy: Vocabulary as sequence of terms

… much more space-efficient than previous scheme



Source: [Introduction to Information Retrieval]()

- Pointers mark the beginning and the end of a vocabulary term.

Vocabulary compression (3)

- Save more space by
  - Grouping k subsequent terms (k-1 pointers are saved per group)
  - Prefix replacement

One block in blocked compression ($k = 4$) …

8automata8automate9automatic10automation

$\Downarrow$

… further compressed with front coding.

8automat∗a1⋄e2⋄ic3⋄ion

Source: [Introduction to Information Retrieval]

Comparison of vocabulary compression strategies

- Compression of vocabulary with ~400,000 terms:

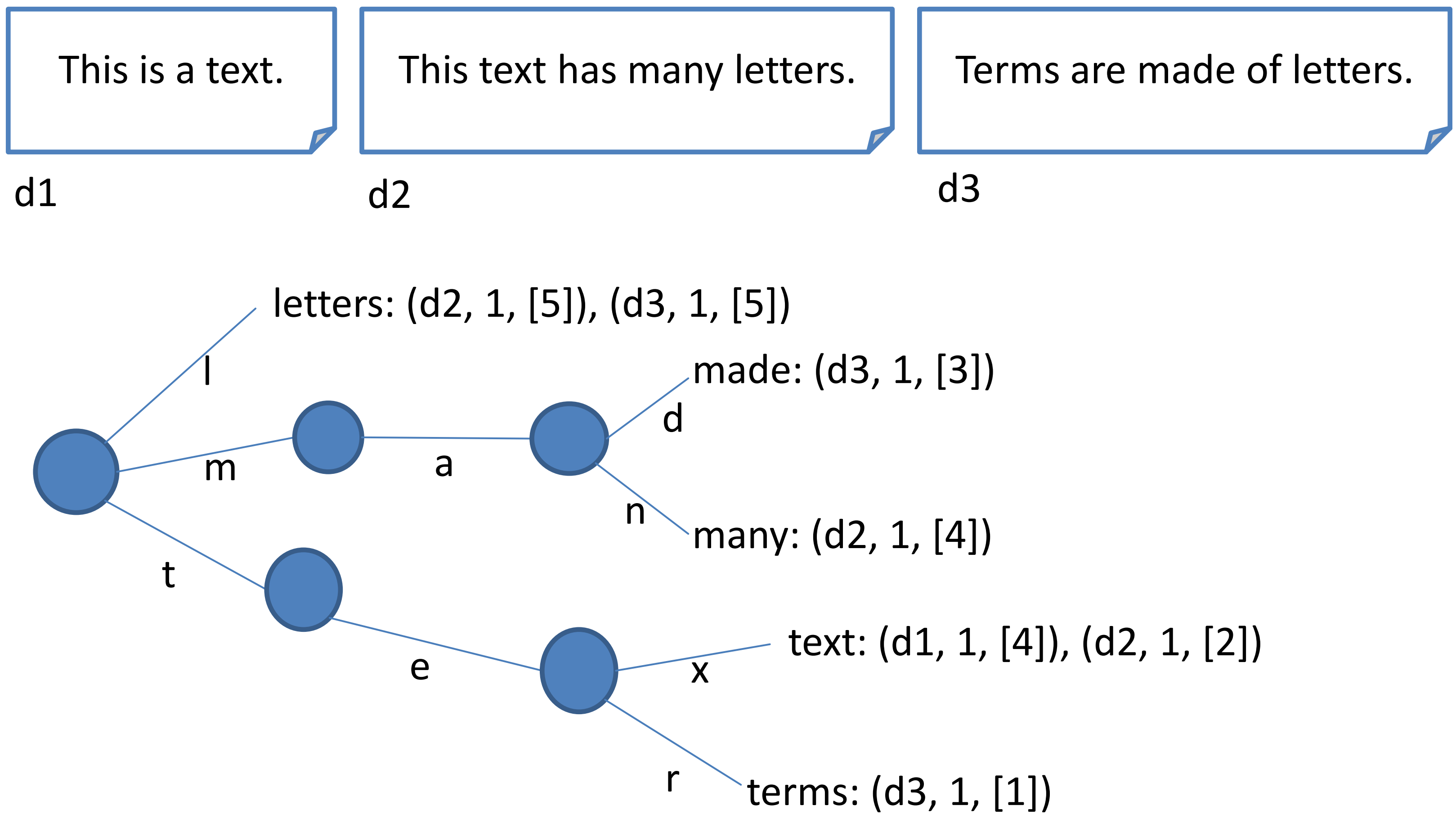
Dictionary compression for Reuters-RCV1.

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
$\sim$, with blocking, $k = 4$	7.1
$\sim$, with blocking & front coding	5.9

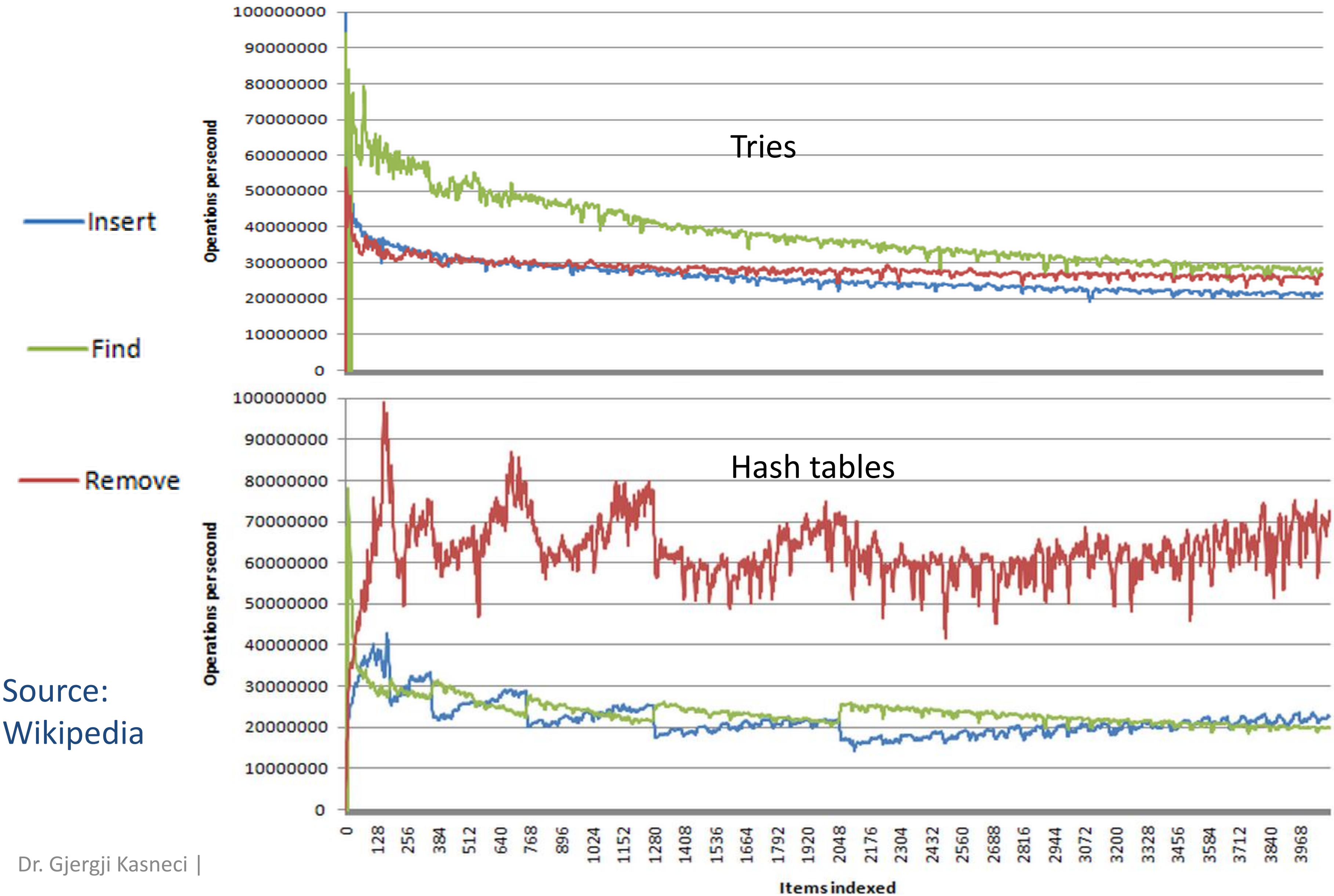
Source: [Introduction to Information Retrieval]

Vocabulary compression with prefix trees

- For vocabularies of moderate size (e.g., for in-memory processable size) use **tries** (conceptually the same as the previous scheme)

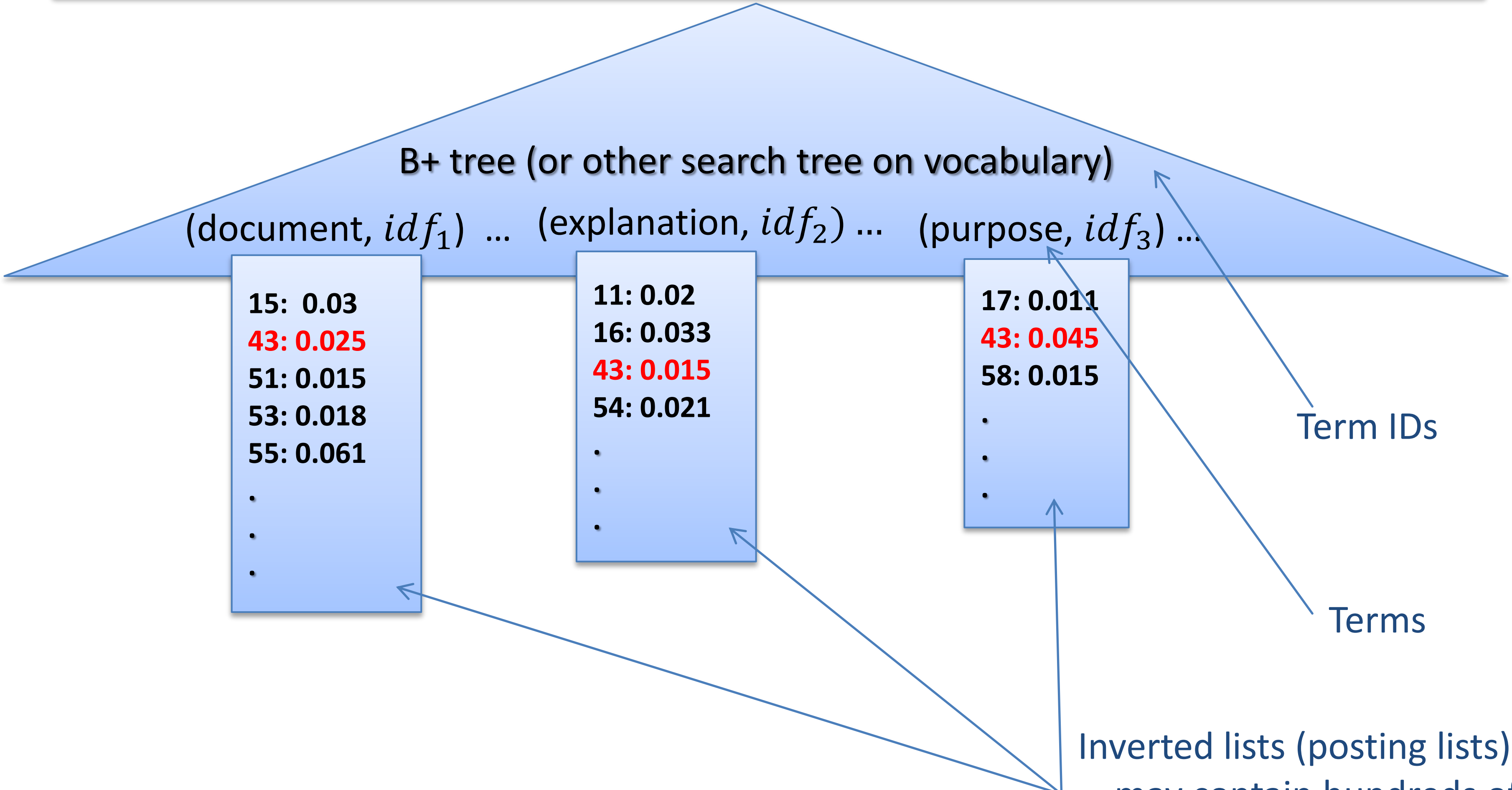


Tries vs. hash tables



Source: Wikipedia

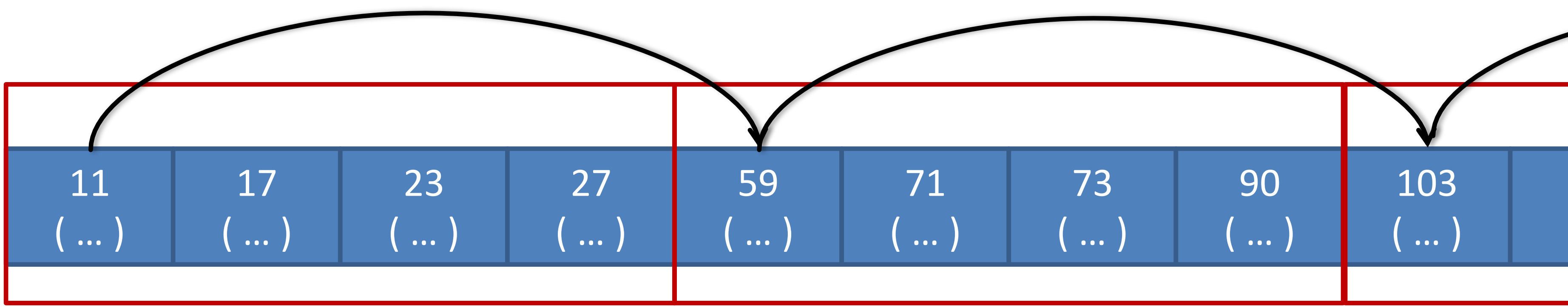
Final Index



- How are the inverted lists stored?

Storing inverted lists

- Partition the list in blocks of same size
- Blocks are stored sequentially
  - We will see later that for Boolean queries sorting by ID is sufficient, for ranking sorting by scores (i.e., term frequencies) is better
- Skip pointers at the beginning of each block point either to the next block or a few blocks ahead



Compressing inverted lists

- Given a Zipf-distribution of terms over the indexed documents, the lengths of the inverted lists will follow the same distribution.

  → Unbalanced latencies for reading lists of highly varying sizes from disk

- Is it possible to mitigate these latencies?

  → Effective compression needed

- Could we apply Ziv-Lempel compression to inverted list entries?

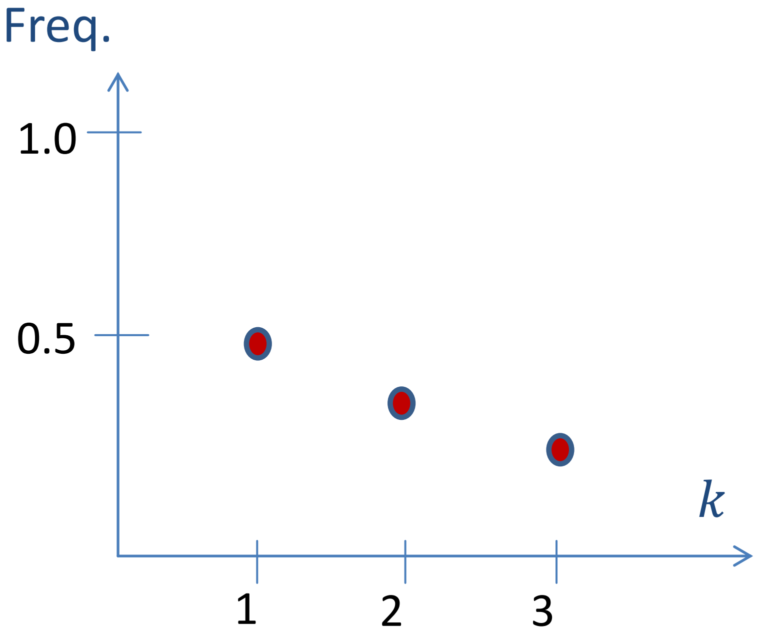
- Ziv-Lempel is good for continuous text but not for postings

- For inverted lists, gaps between successive doc IDs are encoded

Unary encoding of gaps

- Gap size k is is encoded by $(k-1)$-times 0 followed by one 1

Decimal	Unary
1	1
2	01
3	001
4	0001
5	00001
6	000001
7	0000001
8	00000001
9	000000001
10	0000000001



- Optimal for $P(\Delta = k) = \left(\frac{1}{2}\right)^k$

Binary encoding of gaps

- Gap size k is encoded by its binary representation

Decimal	Unary	Binary
1	1	1
2	01	10
3	001	011
4	0001	100
5	00001	101
6	000001	110
7	0000001	111
8	00000001	1000
9	000000001	1001
10	0000000001	1010

- Good for long gaps (but not prefix-free)

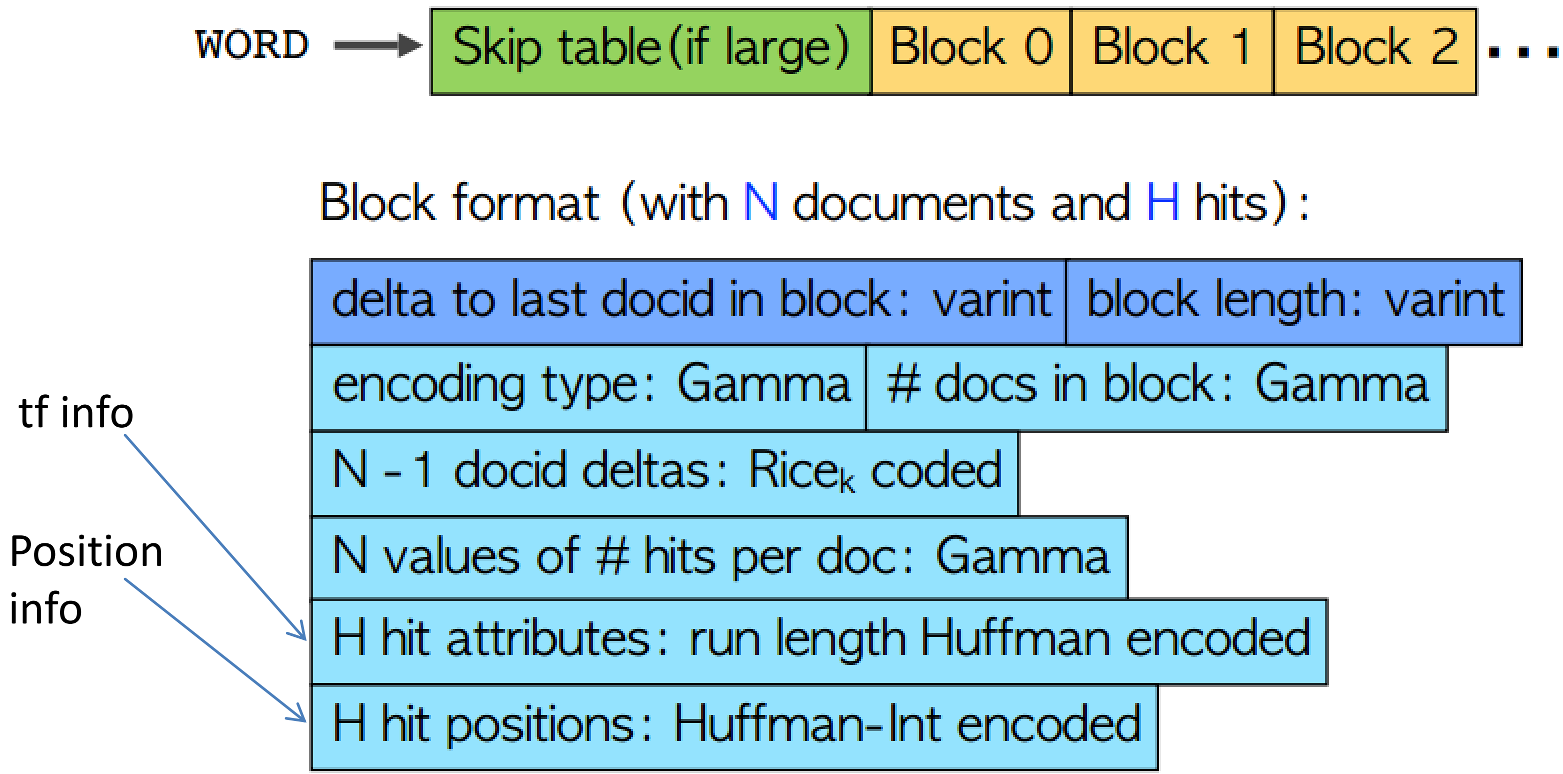
Elias Gamma encoding of gaps

- Gap size k is encoded by $1 + \lfloor \log_2 k \rfloor$ in unary followed by binary representation, without the most significant bit
- E.g.: 9 → 0001 001

Decimal	Unary	Binary	Gamma
1	1	1	1
2	01	10	01 0
3	001	011	01 1
4	0001	100	001 00
5	00001	101	001 01
6	000001	110	001 10
7	0000001	111	001 11
8	00000001	1000	0001 000
9	000000001	1001	0001 001
10	0000000001	1010	0001 010

- Optimal for $P(\Delta = k) \approx \frac{1}{2k^2}$

Google's Gamma encoding scheme



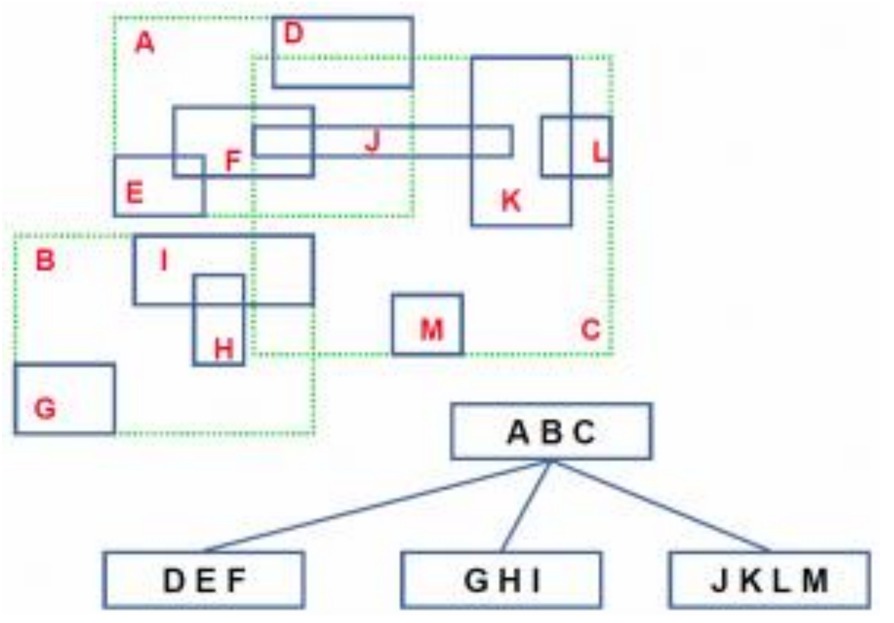
Source: [WSDM 2009 keynote by J. Dean]

Other types of indeces

- Suffix trees

- Index for regular expression queries (e.g. Permuterm Index for wildcard queries)

- R+ trees for spatial data



- Index with temporal information (for temporal queries)

- …

Summary

- Steps to index construction
  - Sorting docs by terms
    → vocabulary construction
    → postings construction
    (Parallelization through MapReduce)
  - Making the vocabulary efficiently searchable with B+ trees
    - Vocabulary compression (sequential term storage with blocking and prefix replacement)
  - Prefix trees for maintaining vocabulary of moderate size in main memory
  - Storing and compressing inverted lists
    - Equal-size blocks with pointers between subsequent blocks
    - Gap-based encoding within blocks (Unary, Gamma, Rice, …)