Inverted Indexing for Information retrieval

Adam Hadraba

Table of contents

Motivation

- **Sequential scan is not efficient**
- **Looking for structure that provides background for:**
  - **Processing large number of queries over massive amount of data each second**
  - **Data set changes**
  - **Different kinds of queries**
  - **Ranking results**
  - **“Fast” response & dealing w/ hardware limitations**

Inverted index

- **Most common indexing method used in IR systems**
- **Way to avoid linearly scanning the texts**
  - **Index in advace**
- **Widely used in search engines**

- **Normally, documents – lists of words**
  - **Inverted index — for each word lists of documents**

Creating Inverted index

1. **Collect documents to be indexed**
2. **Tokenize the text**
3. **Preprocessing**
4. **Indexing**

Dictionary	
term	**doc. freq**
Seminář	20
Laboratoř	15
DISA	33

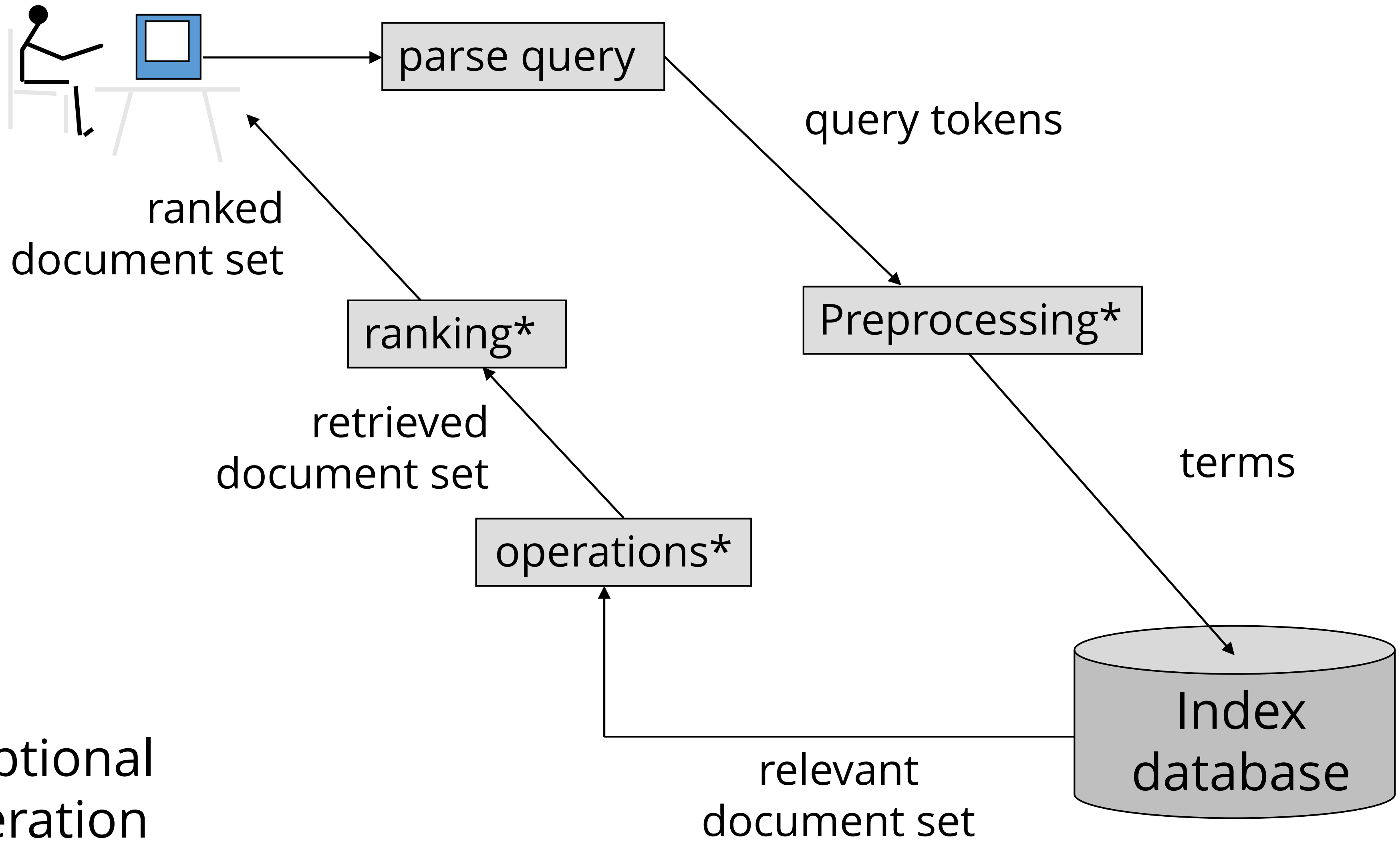
Postings				
1;2	3;5	5;1	6;2	
1;5	2;5	5;3	10;1	...
2;3	4;1	5;1	7;8	

Size

- **Dictionary:**
  - *Heap's law:* V = O(n^{β}), 0.4 < β < 0.6
  - *TREC – 2: 1GB text, 5MB dictionary*
- **Postings**
  - Worst case – one per ocurrence of a word in a text: O(n)
- **Inverted index are big – typically 10-100% the size of collection of documents**

► **Most of the time compression is needed**

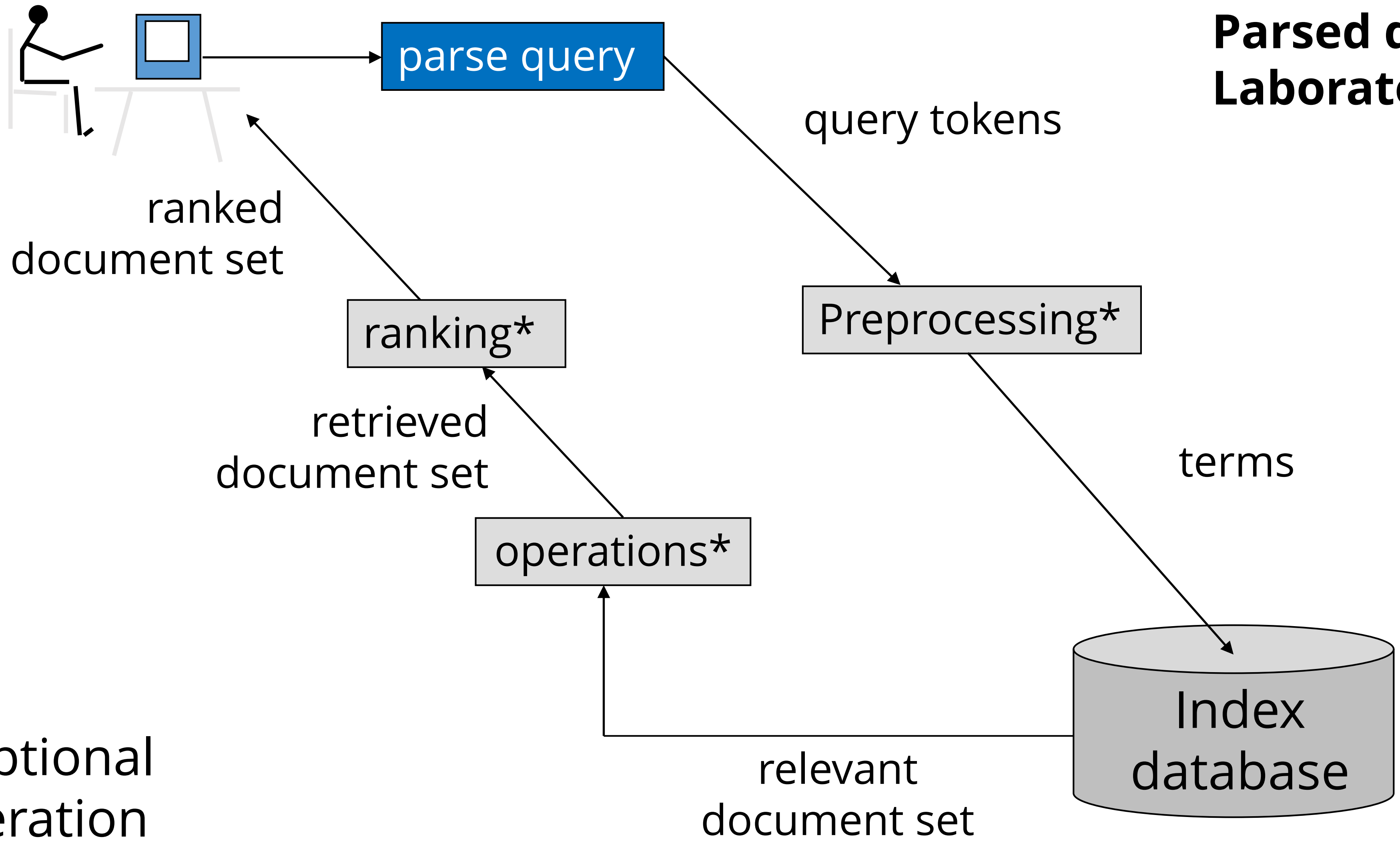
Search subsystem



* optional operation

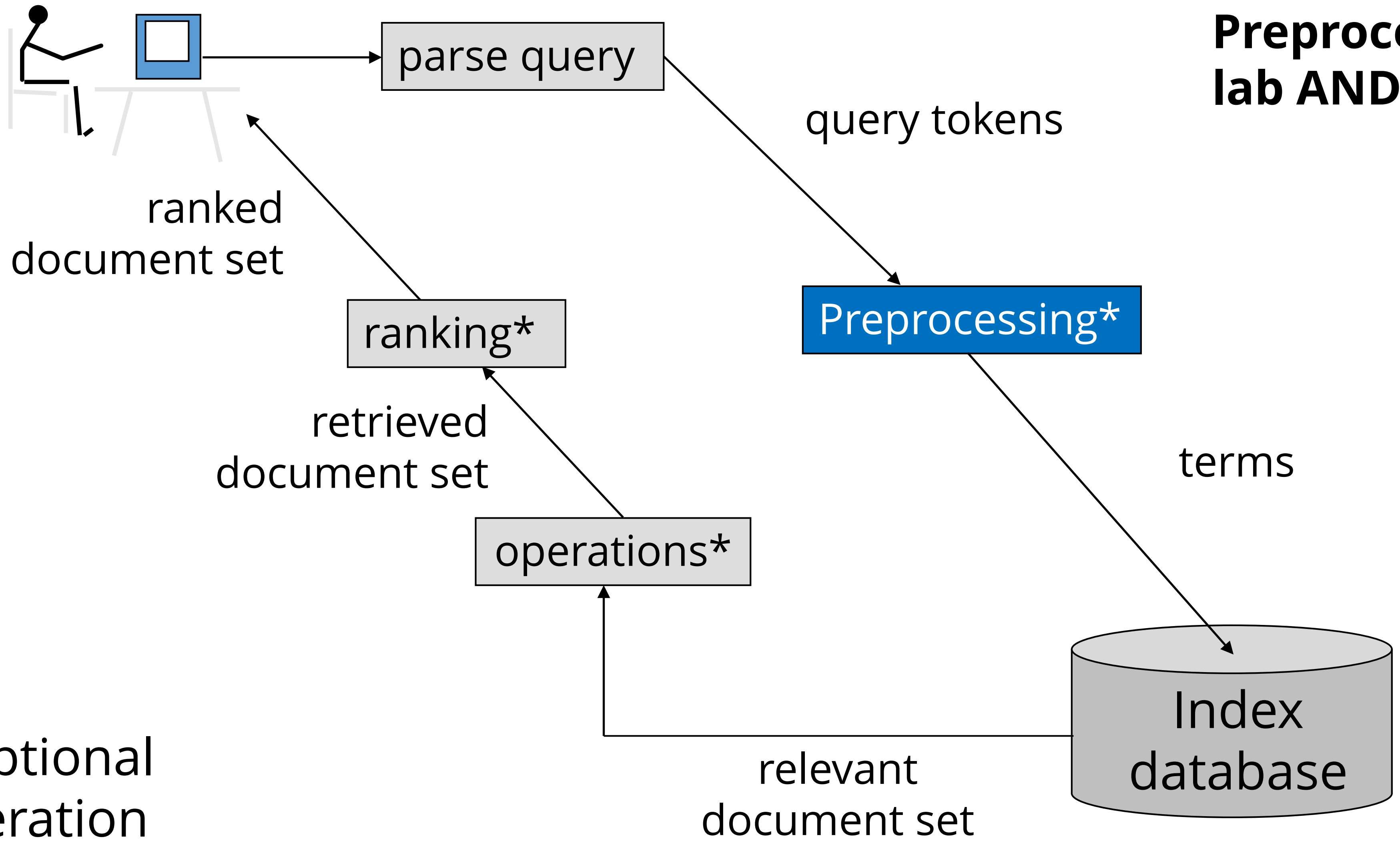
Search subsystem

Parsed query:
Laboratoř AND DISA



* optional operation

Search subsystem



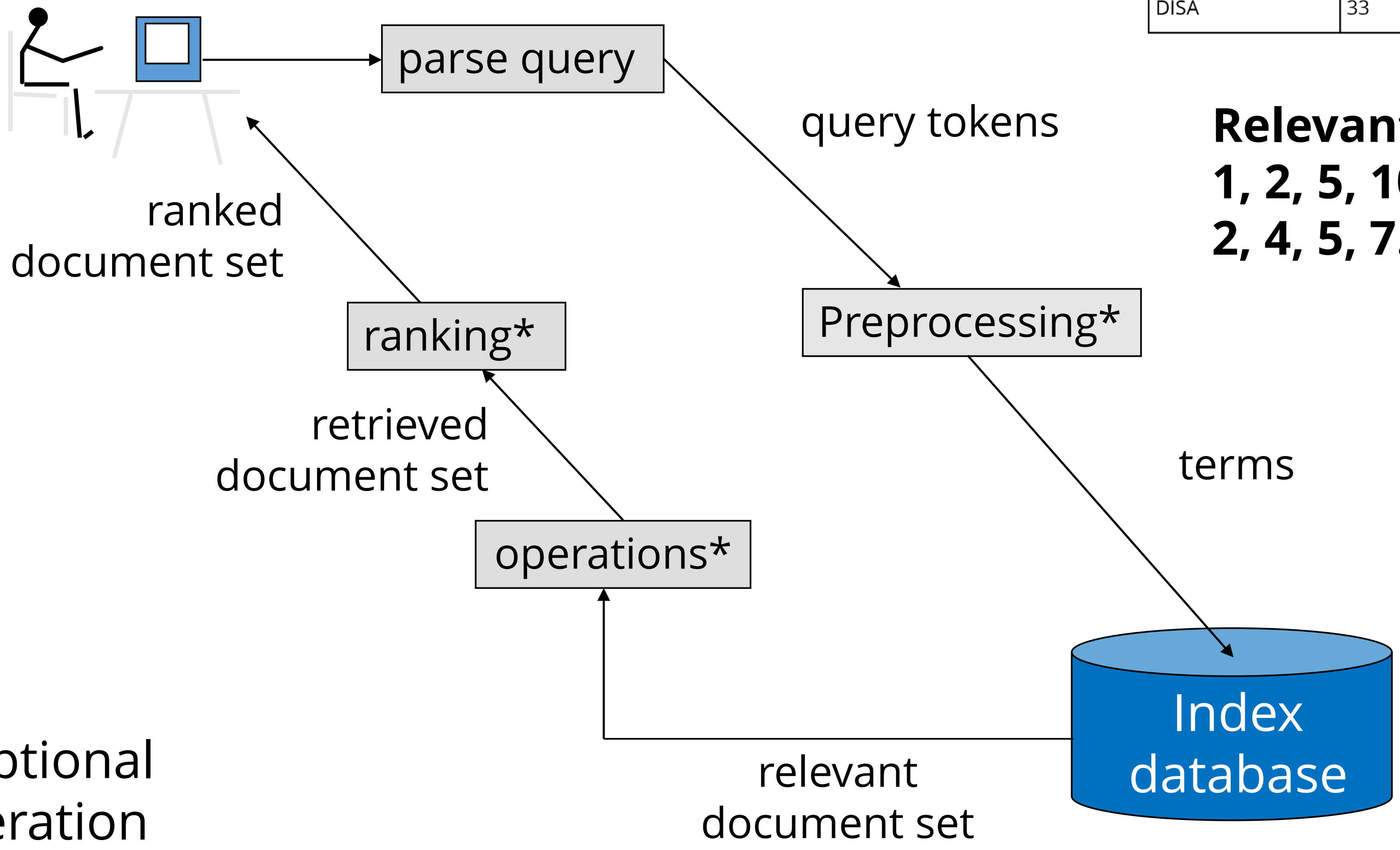
Preprocessed query:
lab AND disa

* optional
operation

Search subsystem

Dictionary	
term	**doc. freq**
Seminář	20
Laboratoř	15
DISA	33

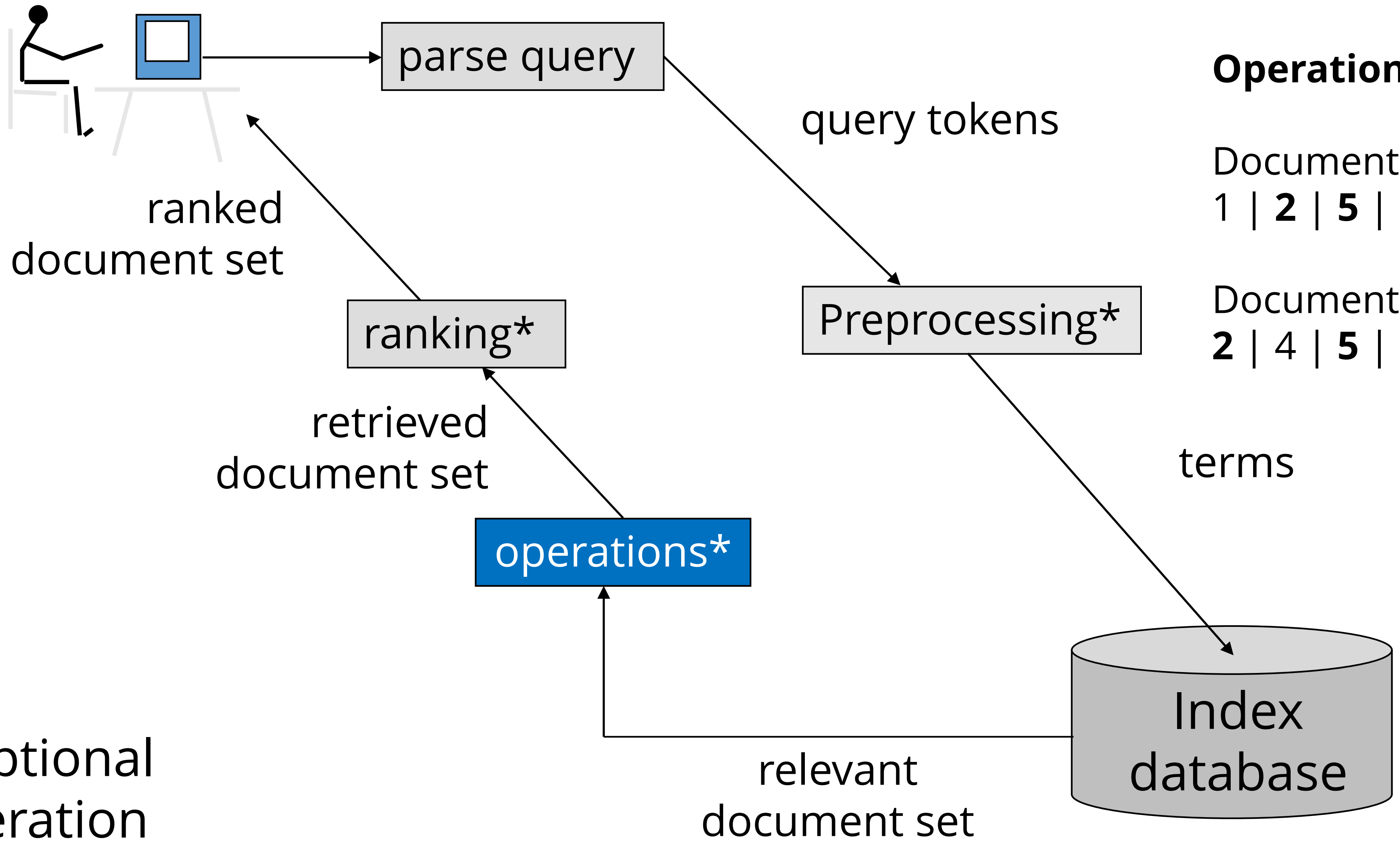
Postings				
1;2	3;5	5;1	6;2	
1;5	2;5	5;3	10;1	…
2;3	4;1	5;1	7;8	



Relevant document set:
1, 2, 5, 10…
2, 4, 5, 7…

* optional operation

Search subsystem



* optional operation

Relevant document set:
1, 2, 5, 10…
2, 4, 5, 7…

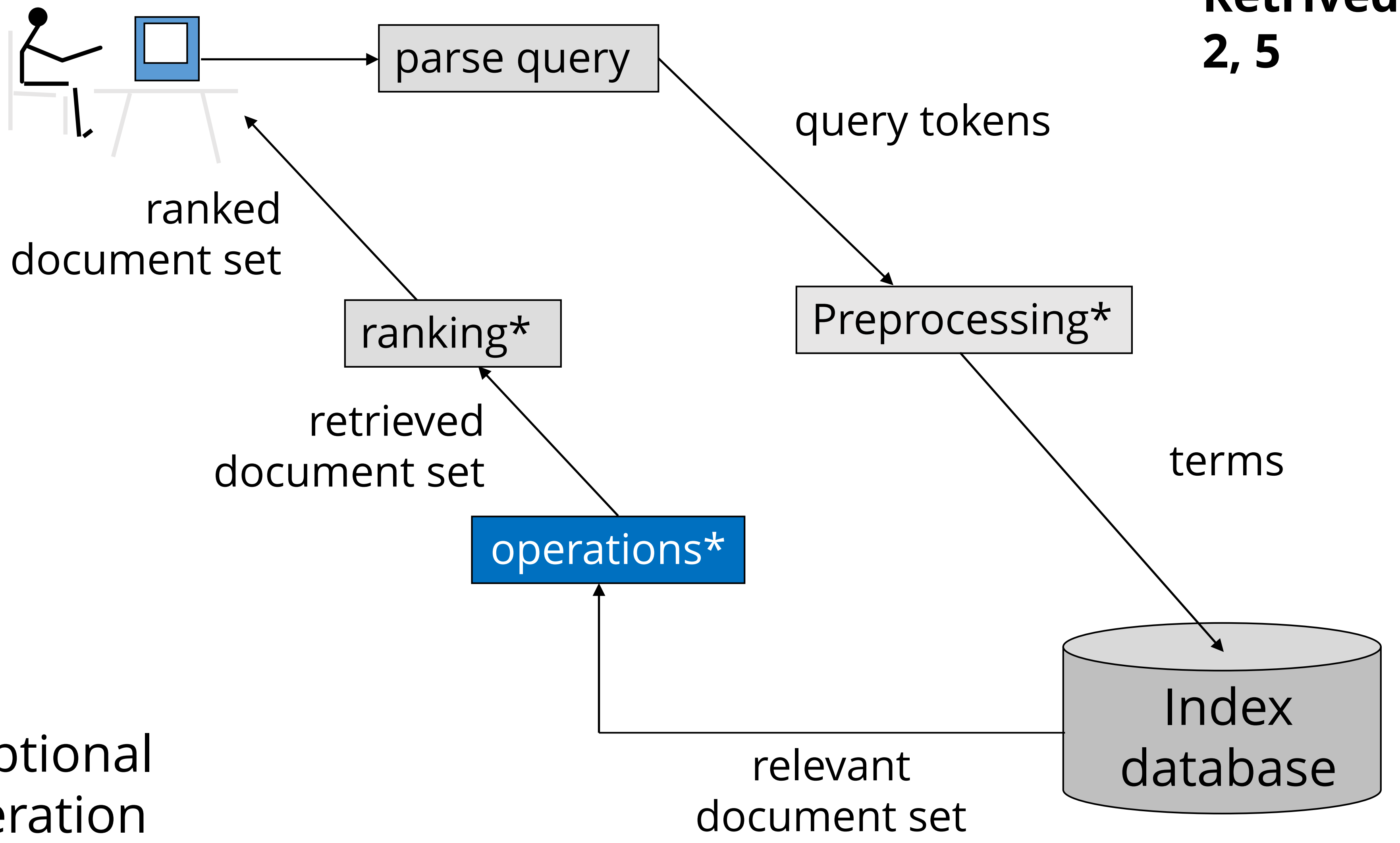
Operation intersection

Documents containing “lab”:
1 | **2** | **5** | 10 …

Documents containing “disa”:
2 | 4 | **5** | 7 …

Search subsystem

Retrived document set:
2, 5



* optional
operation

Not simple as that

- **Support for different queries:**
  - **Boolean, proximity, phrase, wildcard…**
    - **More complex postings**
    - **Additional indexes**

    ► increase in complexity
- **Retrieved document set could be huge**
  - **We need to rank them relevantly**

Vector space model

Vector space model

- Idea: A user‘s query can be viewed as a short document
- Documents and queries are represented as vectors in term space (both in the same space)

► We are able to measure proximity — rank retrieved documents

Vector space model cont.

- **Two documents are similar, if they contain some of the same terms.**
- **We can take into account / weighting:**
  - Lenght of documents
  - Number of terms in common
  - Unusual or common words
  - How many times each term appears

- **Documents are represented as "bag of words"**
  - Words are terms with no order
  - ► Thus the document

    John is quicker than Mary.

    Is indistinguishable from

    Mary is quicker than John.

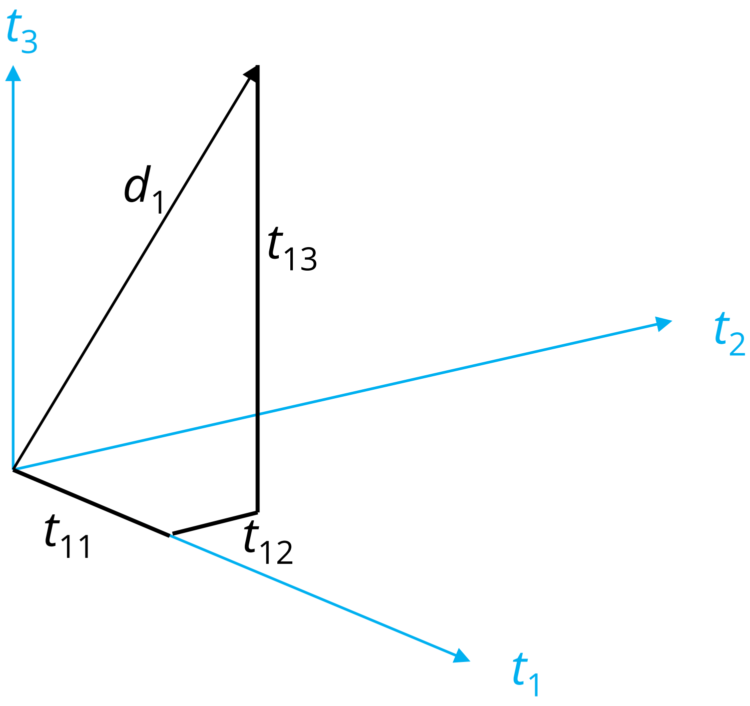
Vector space model cont.

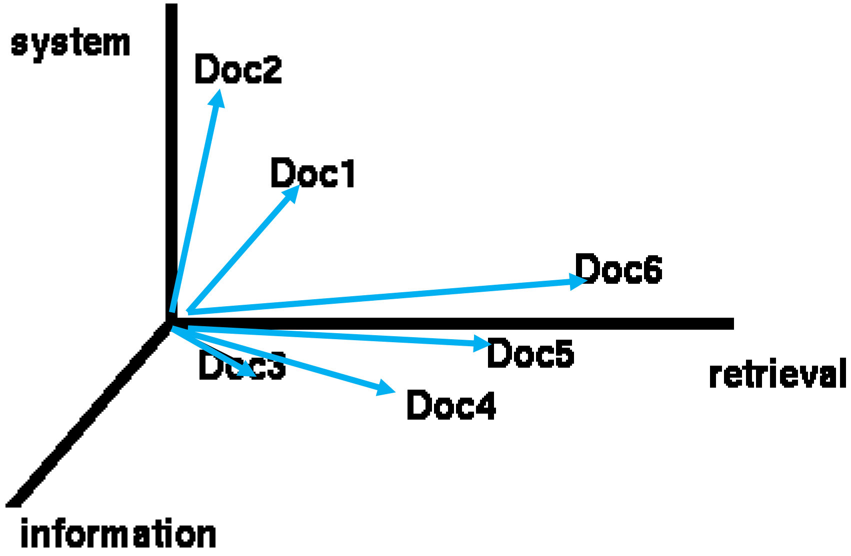
- *Term vector space*
  - *n*-dimensional space
  - *n* – number of different terms/tokens used to index a set of documents
- Vector
  - Document *i*, d_i, represented by a vector. Its magnitude in dimension *j* is w_{ij}, where:

$$w_{ij} > 0 \qquad \textit{if term j occurs in document i}$$
$$w_{ij} = 0 \qquad \text{otherwise}$$

  - w_{ij} is the weight of term *j* in document *i*.

Documents in 3-dimensional term vector space





Assumption: Documents that are "close together" in space are closer in meaning

Measuring similarity

- Eg. **Cosine angle** between the docs $\mathbf{d}_1$ and $\mathbf{d}_2$ determines doc similarity

$$\cos(\theta) = \frac{\mathbf{d}_1 . \mathbf{d}_2}{|\mathbf{d}_1| \, |\mathbf{d}_2|}$$

$\cos(\theta) = 1$ – *documents exactly the same;*

$\cos(\theta) = 0$ – *totally different.*

Constructing inverted index

Hardware dependencies

- Memory is faster than disc
  - Seek time, transfer from disc
  - ► As much data as possible in memory
- Better with SSD
- Disc to memory handled by system bus, not processor
  - Reading compressed data and uncompressing
    usually faster than
    reading uncompressed data

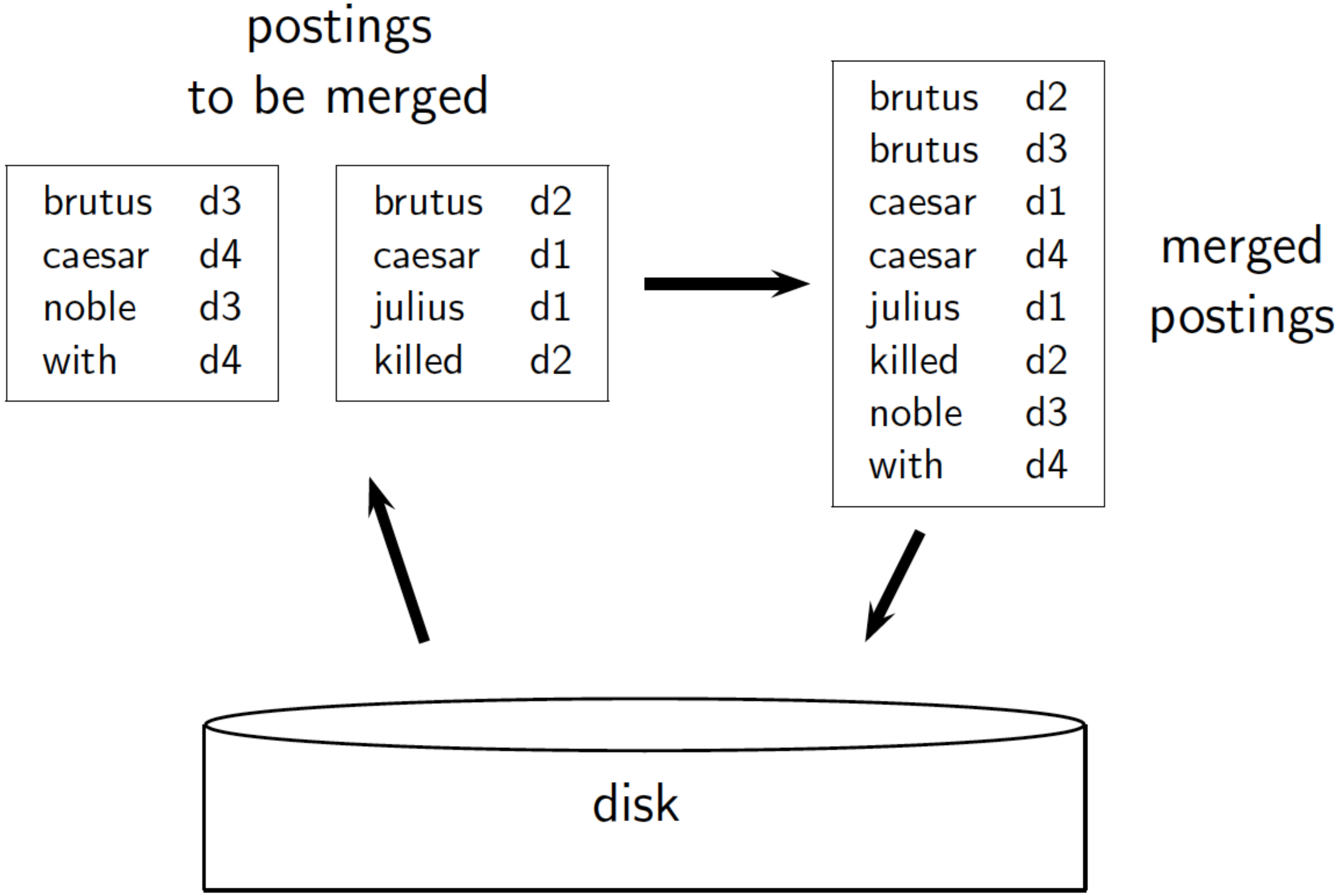
Blocked Sort-Based Indexing (BSBI)

- **Memory is insufficient, we need to use disc**
- **Map term to termID**

1. **Divide documents collection into blocks**
   - **Each block fits into main memory**
2. **For each block**
   - **Sort the termID-docID pairs**
   - **Store intermediate sorted result on disc**
3. **Merge all intermediate results into the final result**
   - **Maintaining small read and write buffers**

- **Assumption: dictionary fits into main memory, termID available online for each document**

Blocked Sort-Based Indexing (BSBI)



Blocked Sort-Based Indexing (BSBI)

- **Problems:**
  - **Dictionary must fit into memory**
  - **We need dictionary to map a term to termID**
  - **term-docID postings instead of termID-docID**
    - **But intermediate files would become very large.**
    - **Scalable, but slow.**

Single Pass In-Memory Indexing (SPIMI)

- Dictionary won‘t fit into memory

1. Dictionary for each block
2. Add a posting directly to its posting list
    - No sorting
    - No storage of termID-docID pairs
    - Posting list doubles allocated space each time it‘s full

    ► Complete inverted index for each block
3. Merge into one big index

- Compression makes SPIMI more efficient
    - Postings
    - Dictionary terms

    ► Processing larger blocks

Distributed indexing

- For web-scale indexing
  - Distributed computer cluster
  - Individual machines are fault-prone

- Maintain a master machine directing the indexing job
- Break up indexing into set of (parallel) tasks
- Master machine assigns tasks

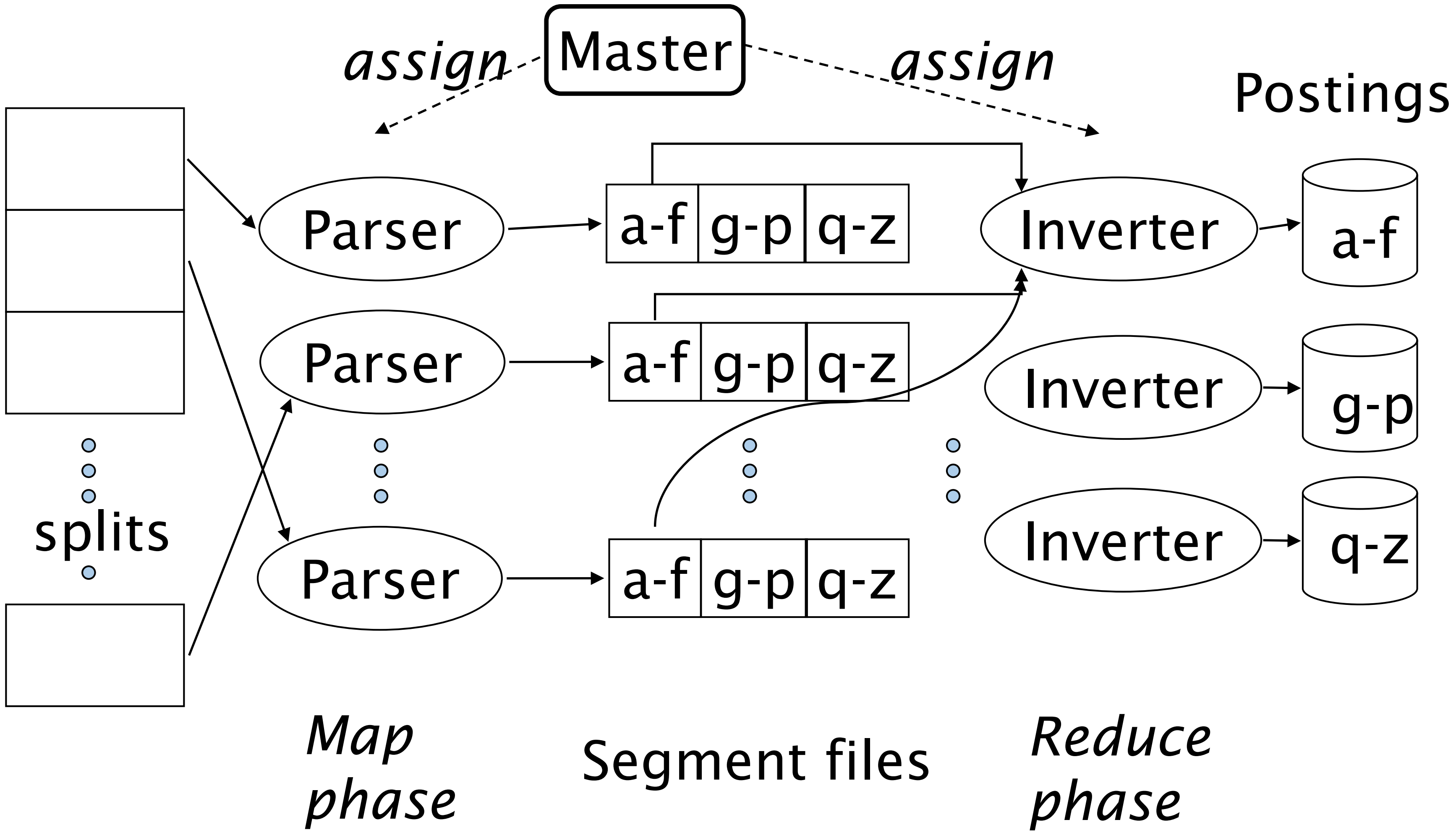
Distributed indexing cont.

- **Two sets of tasks**
  - Parsers
  - Inventers
- **Braking the input documents into splits (corresponding to blocks in BSBI/SPIMI)**

Distributed indexing

- **Parsers**
  - **Master assign split to an idle parser machine**
  - **Parser reads a document and emits term-doc pairs**
  - **Parser writes pairs into *j* partitions**
  - **Each partition is for a range terms' first letter**
    - (e.g. **a-f**, **g-p**, **q-z**) — here *j*=3
- **Inverter**
  - **Collects all term-doc pairs from one term-partition**
  - **Sorts and writes to postings lists**

Distributed indexing – data flow



Dynamic indexing

- **Untill now, we assumed that collections are static**

- **New documents need to be iserted**
- **Documents are deleted and modified**

► **Postings upades for terms already in dictionary**

► **New terms added to dictionary**

Dynamic indexing

- **“Big” main index**
- **New documents go into „small“ auxiliary index**
- **Search across both, merge results**
- **Deletions**
  - Invalidation bit-vector
- **Periodically, re-index into one main index**

Dynamic indexing

- **Problems:**
  - **Poor performance during merge**
  - **If we have separate files for each postings list, merging is efficient (simple append)**
  - **Lots of files — not efficient for O/S**
- **In reality: somewhere in between**
  - **Split large postings lists**
  - **Collect postings list of lenght 1 in one file etc.**

We covered

- **Structure of inverted index**
- **Ranking – Vector Space Model**
- **Constructing of inverted indexes**

Thank You.