Advanced Topics in Information Retrieval

3. Efficiency & Scalability

Vinay Setty
(vsetty@mpi-inf.mpg.de)

Jannik Strötgen
(jtroetge@mpi-inf.mpg.de)

max planck institut
informatik

MAX-PLANCK-GESELLSCHAFT

Outline

3.1. Motivation

3.2. Index Construction & Maintenance

3.3. Static Index Pruning

3.4. Document Reordering

3.5. Query Processing

Outline

3.1. Motivation

3.2. Index Construction & Maintenance

3.3. Static Index Pruning

3.4. Document Reordering

3.5. Query Processing

3.1. Motivation

- **Efficiency** is about *"doing things right"*, i.e., accomplishing a task using minimal resources (e.g., CPU, memory, disk)

- **Scalability** is about to be able to
  - accomplish a larger instance of a task e.g. indexing millions/ billions of documents, large number of queries
  - using additional resources (e.g., faster/more CPUs, more memory/disk)

Indexing & Query Processing

- Our focus will be on **two major aspects** of every IR system
  - **indexing**: how can we efficiently **construct & maintain** an inverted index that consumes **little space**
  - **query processing**: how can we **efficiently identify the top-*k* results** for a given query without having to read posting lists completely

- Other aspects which we will not cover include
  - **caching** (e.g., posting lists, query results, snippets)
  - **modern hardware** (e.g., GPU query processing, SIMD compression)

Hardware & Software Trends

- CPU speed has increased more than that of disk and memory: **faster to read & decompress** than to read uncompressed

- More memory is available; disks have become larger but not faster: now common to keep **indexes in (distributed) memory**

- Many (less powerful) instead of few (powerful) machines; platforms for **distributed data processing** (e.g., MapReduce, Spark)

- More **CPU cores** instead of faster CPUs; **SSDs** (fast reads, slow writes, wear out) in addition to HDDs; **GPUs** and **FPGAs**

Outline

3.1. Motivation

3.2. Index Construction & Maintenance

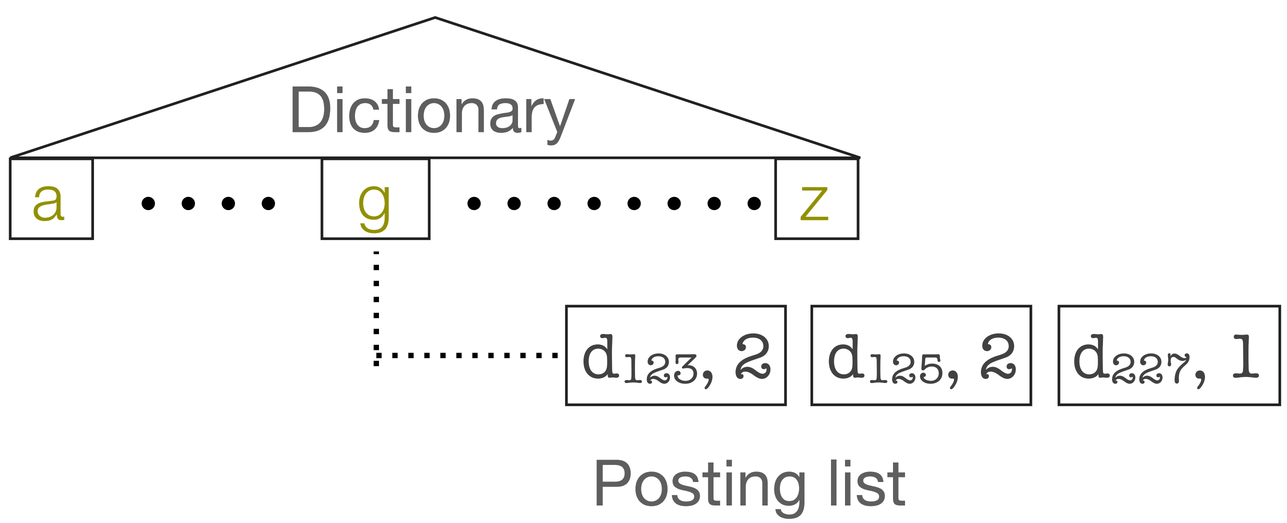
3.3. Static Index Pruning

3.4. Document Reordering

3.5. Query Processing

3.2. Index Construction & Maintenance

- **Inverted index** as widely used index structure in IR consists of
  - **dictionary** mapping terms to term identifiers and statistics (e.g., idf)
  - **posting lists** for every term recording details about its occurrences



- *How to construct an inverted index from a document collection?*
- *How to maintain an inverted index as documents are inserted, modified, or deleted?*

max planck institut informatik

Index Construction

- Observation: Constructing an inverted index (aka. inversion) can be seen as **sorting a large number of** (term, did, tf) **tuples**
    - seen in (did)-order when processing documents
    - needed in (term, did)-order for the inverted index
- Typically, the set of all (term, did, tf) tuples does **not fit into the main memory** of a single machine, so that we need to **sort using external memory** (e.g., hard-disk drives)

Index Construction on a Single Machine

- Lester al. [5] describe the following algorithm by Heinz and Zobel to construct an inverted index on a single machine
  - let B be the number of (term, did, tf) tuples that fit into **main memory**
  - **while** not all documents have been processed
    - **read** (up to) B tuples from the input (documents)
    - **construct in-memory inverted** index by grouping & sorting the tuples
    - **write in-memory inverted index** as sorted run of (term, did, tf) tuples **to disk**
  - **merge on-disk runs** to obtain global inverted index

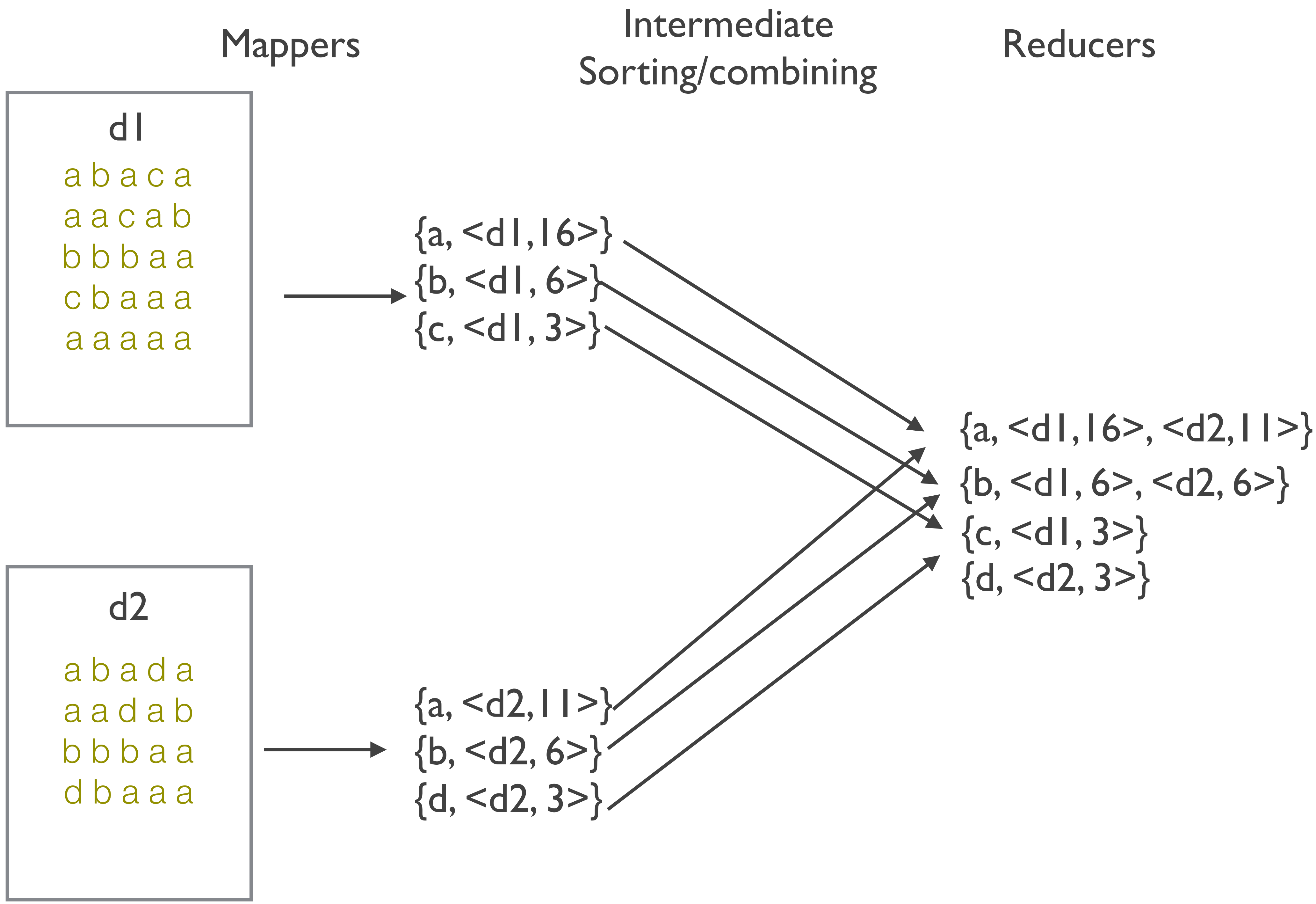
max planck institut informatik

Index Construction in MapReduce

- **MapReduce** as a platform for **distributed data processing**
  - was developed at **Google**
  - operates on large clusters of **commodity hardware**
  - handles **hard- and software failures** transparently
  - open-source implementations (e.g., **Apache Hadoop**) available
  - programming model operates on **key-value (kv) pairs**
  - **map()** reads input data (k_1,v_1) and emits kv pairs (k_2,v_2)
  - platform groups and sorts kv pairs (k_2,v_2) automatically
  - **reduce()** sees kv pairs $(k_2, list<v_2>)$ and emits kv pairs (k_3,v_3)

max planck institut
informatik

Map/Reduce Example



Index Construction in MapReduce

```
map(did, list<term>)
    map<term, integer> tfs = new map<term, integer>();
    // determine term frequencies
    for each term in list<term>:
        tfs.adjustCount(term, +1);
    // emit postings
    for each term in tfs.keys():
        emit (term, (did, tfs.get(term)));

    // platform groups & sorts output of map phase by term

reduce(term, list<(did, tf)>)
    // emit posting list
    emit (term, list<(did, tf)>)
```

max planck institut
informatik

Index Maintenance

- Document collections are **not static**, but documents are **inserted, modified, or deleted** as time passes; changes to the document collection should **quickly be visible in search results**

- Typical approach: Collect changes in main memory
  - **deletion list** of deleted documents
  - **in-memory delta inverted index** of inserted and modified documents
  - **process queries over both** the on-disk global and in-memory delta inverted index and **filter out** result documents from the deletion list

- *What if the available main memory has been exhausted?*

Rebuild

- Rebuild the on-disk global index from scratch
  - in a separate location; switch over to new index once completed
  - attractive for small document collections
  - attractive when document deletions are common
  - requires re-processing of entire document collection
  - easy to implement

Merge

- **Merge** the on-disk global index with the in-memory delta index
  - in a **separate location**; switch over to new index once completed
  - for each term, **read** posting lists from on-disk global index and in-memory delta index, **merge** them, **filter out** deleted documents, and **write** the merged posting list to disk
  - requires **reading entire on-disk global index**

- <u>Analysis</u>: Let B be capacity of the in-memory delta index (in terms of postings) and N be the total number of postings
  - N / B merge operations each having cost $O(N)$
  - total cost is in $O(N^2)$

Geometric Merge

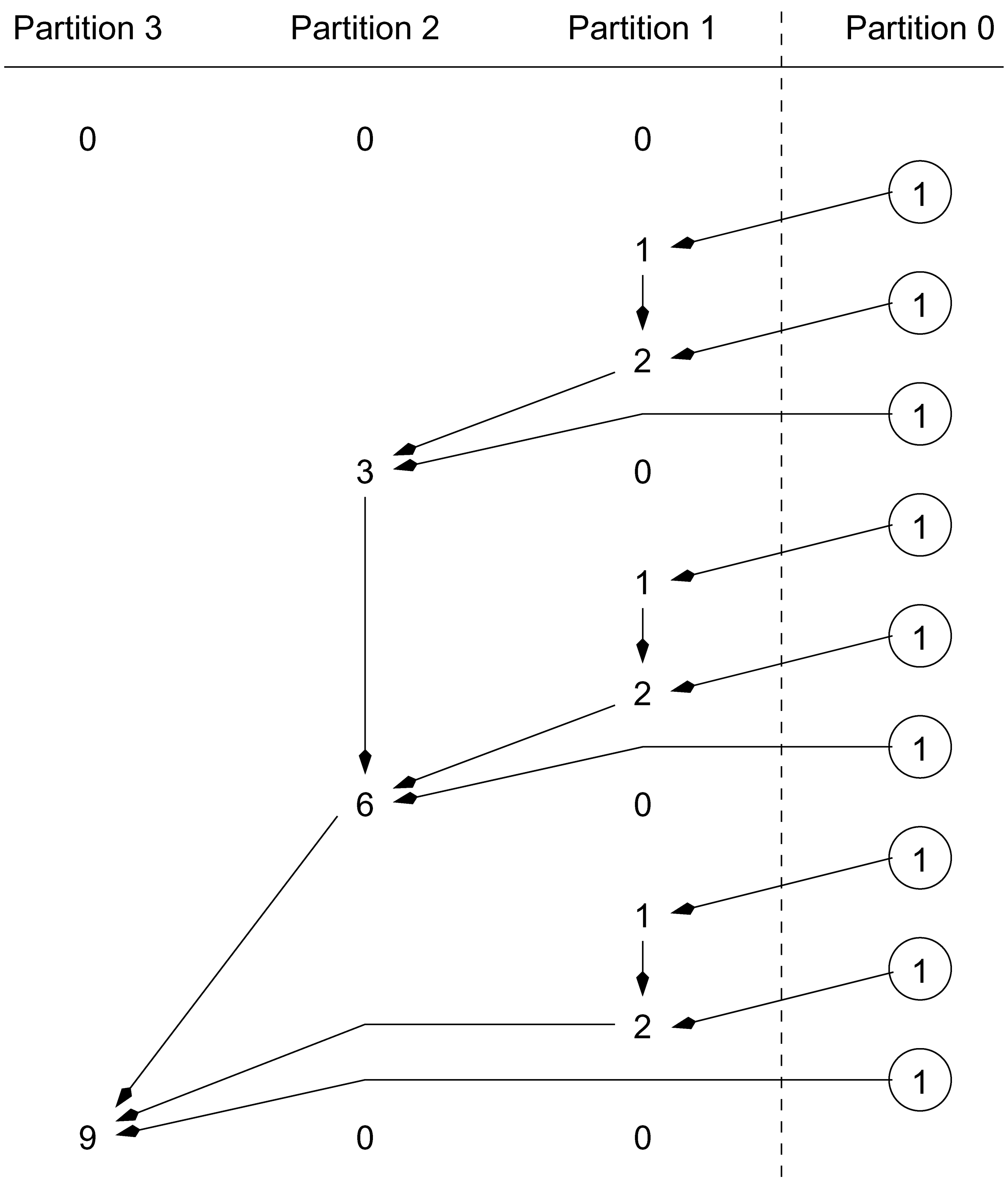
- Lester et al. [5] propose to **partition the inverted index** into index partitions of geometrically increasing sizes
  - tunable by parameter r
  - index partition P_0 is in **main memory** and contains up to B postings
  - index partitions P_1, P_2, … are **on disk** with capacity invariants
    - partition P_j contains at most $(r-1)\ r^{(j-1)}$ B postings
    - partition P_j is either empty or contains at least $r^{(j-1)}$ B postings
  - whenever P_0 **overflows**, a **merge** is triggered

- **Query processing** has to access all (non-empty) partitions P_i, leading to **higher cost** due to required disk seeks

max planck institut informatik

Geometric Merge

r=3



Geometric Merge

- Analysis: Let B be the capacity of the in-memory partition P_0 and N be the total number of postings
  - there are at most $1 + \lceil \log_r(N/B) \rceil$ partitions
  - each posting merged at most once into each partition
  - total cost is $O(N \log N/B)$

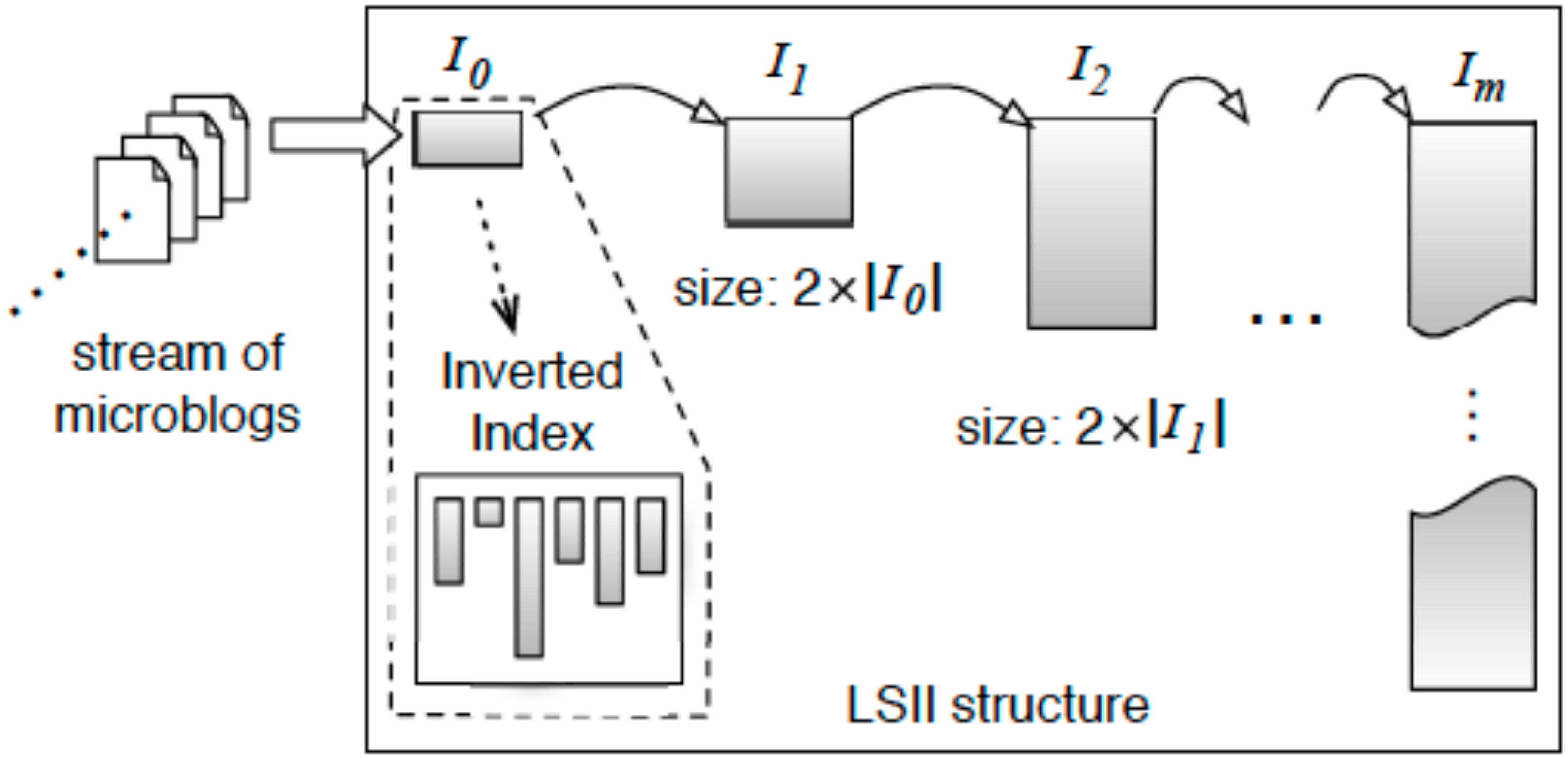
Logarithmic Merge

- **Logarithmic merge** is a simplified variant of geometric merge
  - partition P_0 is in **main memory** and contains B postings
  - partition P_1 is on disk and contains up to **2B** postings
  - partition P_2 is on disk and contains up to 4B postings
  - partition P_j is on disk and contains up to 2^jB postings
  - whenever P_0 overflows, a cascade of merges is triggered
- **Log-structured merge tree** (LSM-Tree) prominent in database systems (e.g., to manage logs) is based on the same principle

max planck institut informatik

Index Maintenance for Microblogs

- Wu et al. [9] use the **log-structured inverted index** to support high update rates when indexing **social media**



Index Management in Elasticsearch

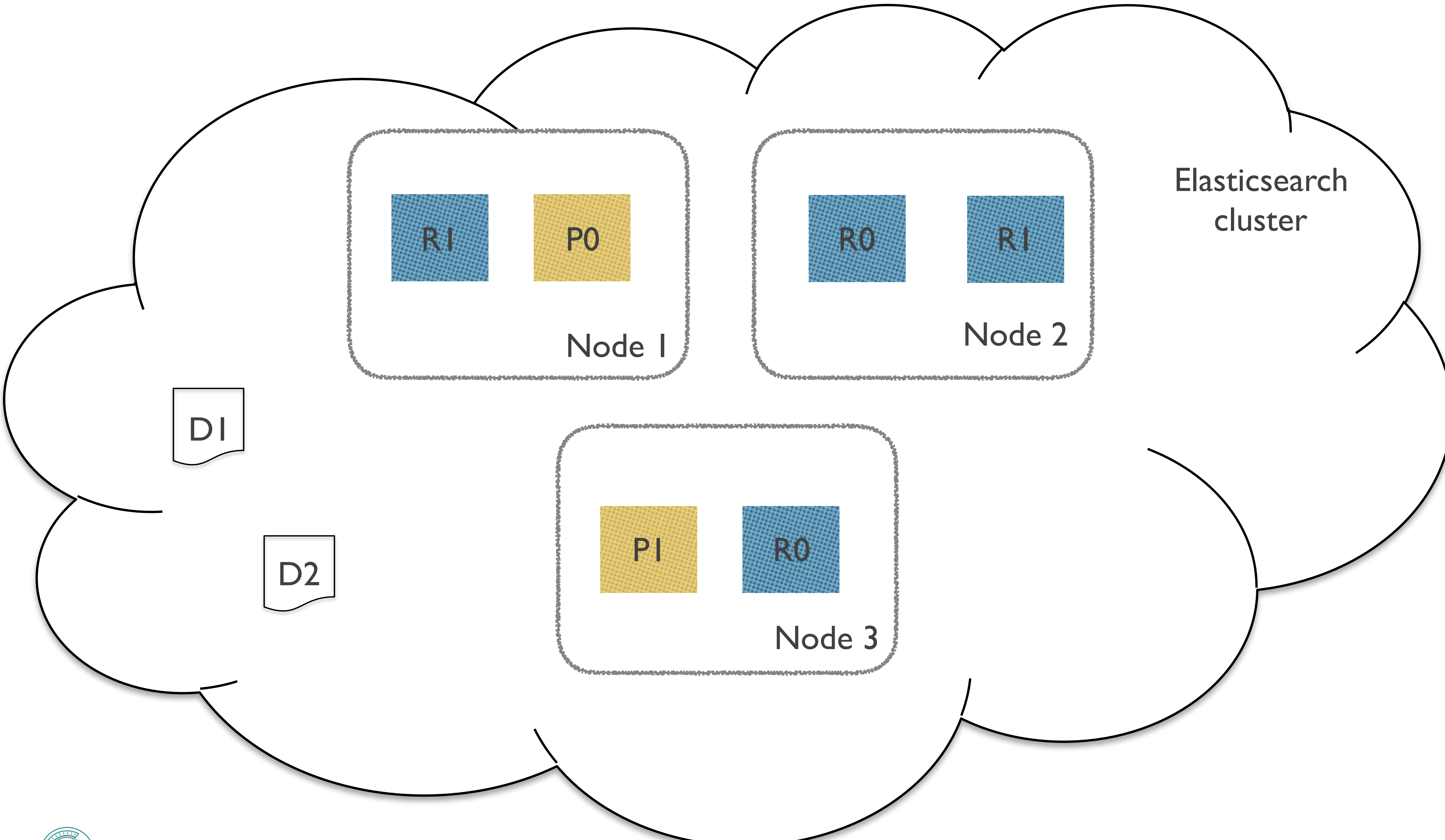
- Indexes are stored as shards
  - Each index has a fixed number of shards
  - By default 5 shards per index - primary shards
- Shards are replicated
  - Each primary shard is replicated
  - Replication factor is a parameter
- Why shards?

A shard is a fully contained horizontal partition of index

Load balance
Distribution
Fault tolerance

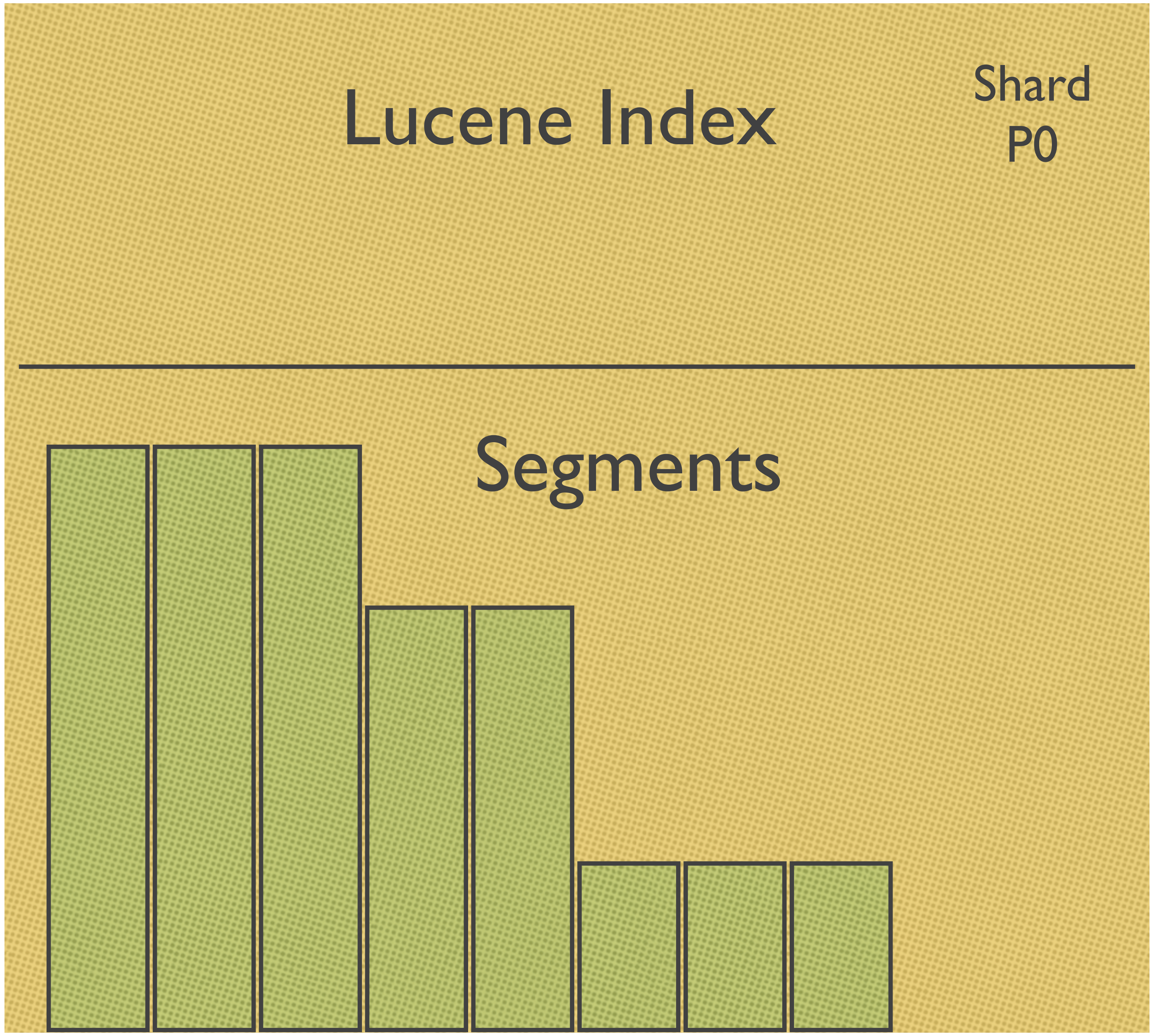
max planck institut informatik

Index Management in Elasticsearch

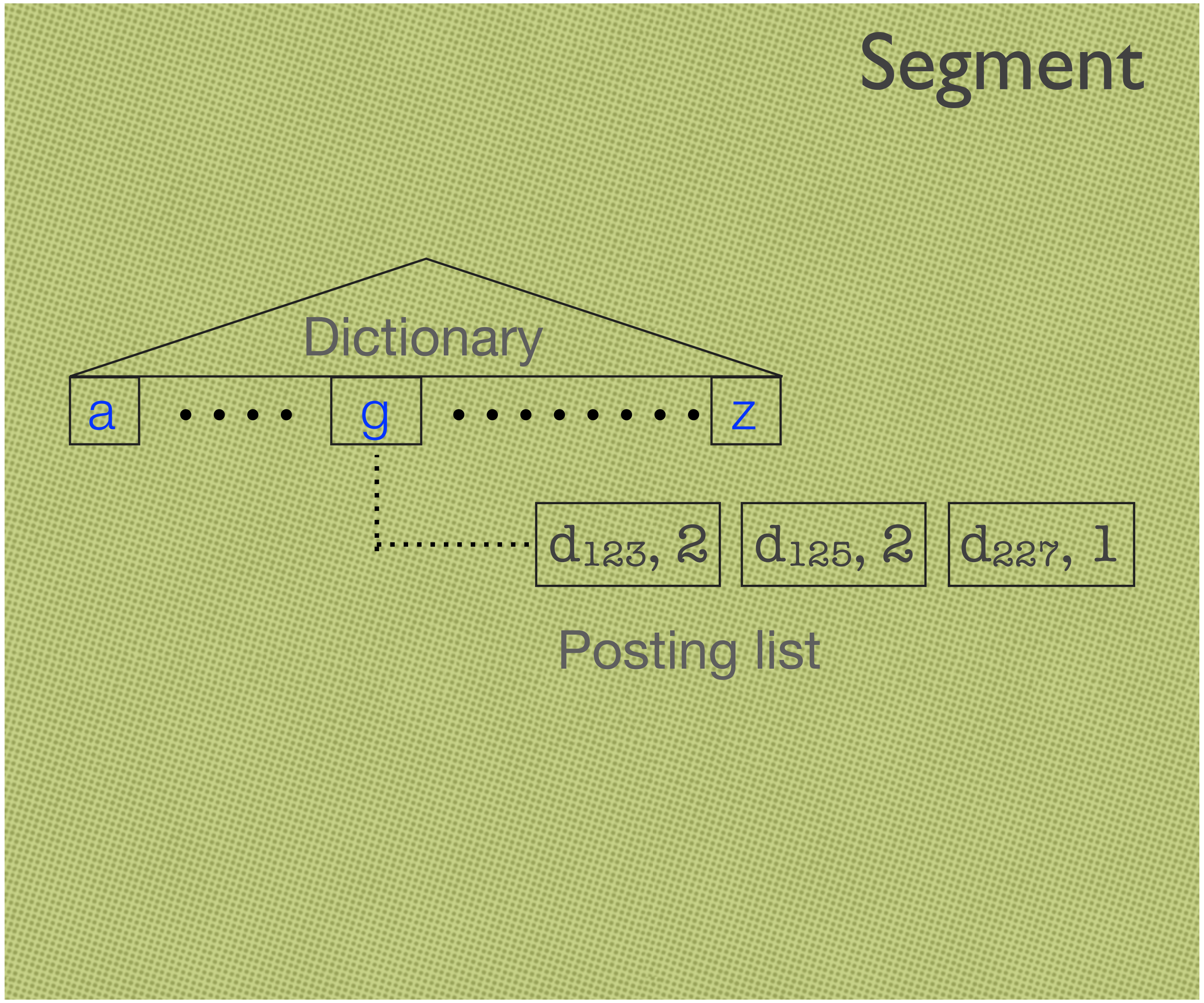


Elasticsearch Shards

- Shards are **immutable**
- Insert only!
- New documents are added to smaller segments
- When segments grow they are merged



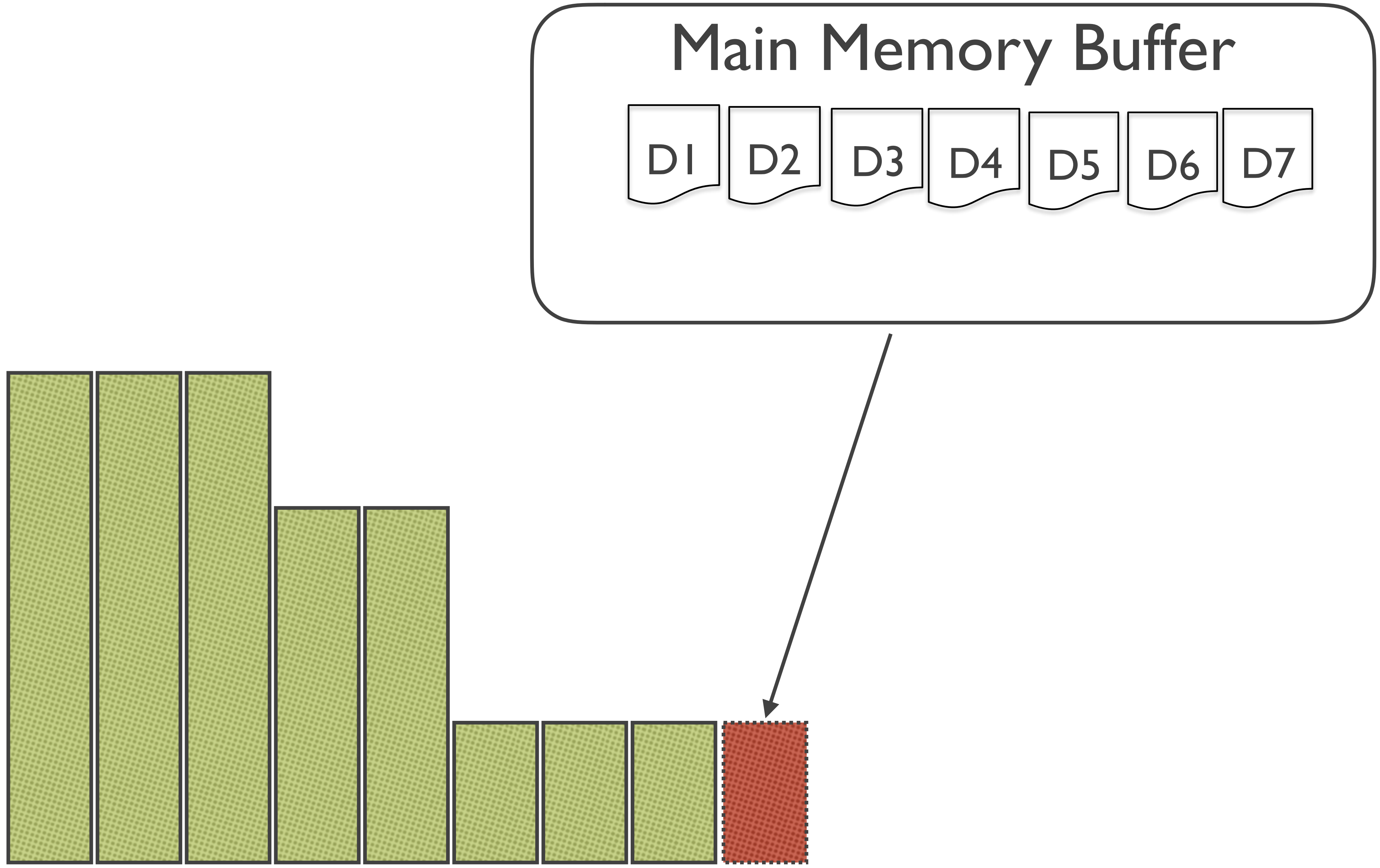
Elasticsearch Shards



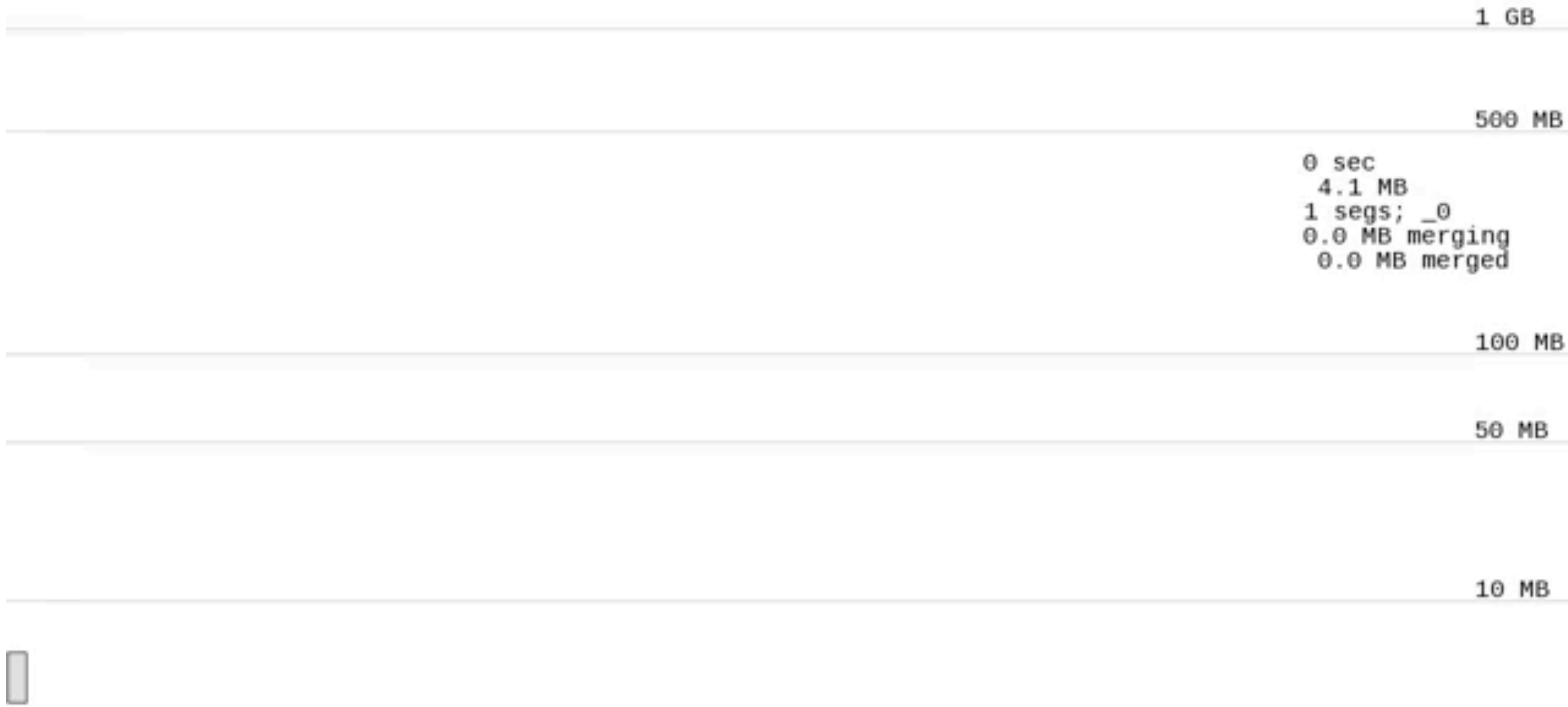
Lucene Dynamic Indexing

- Segments in Lucene are immutable
  - Cannot be changed
  - Can be created, merged and deleted
- When new documents are added
  - Small segments are created
  - When number of segments grow
  - Some merging technique is used such as logarithmic merging

Dynamic Indexing

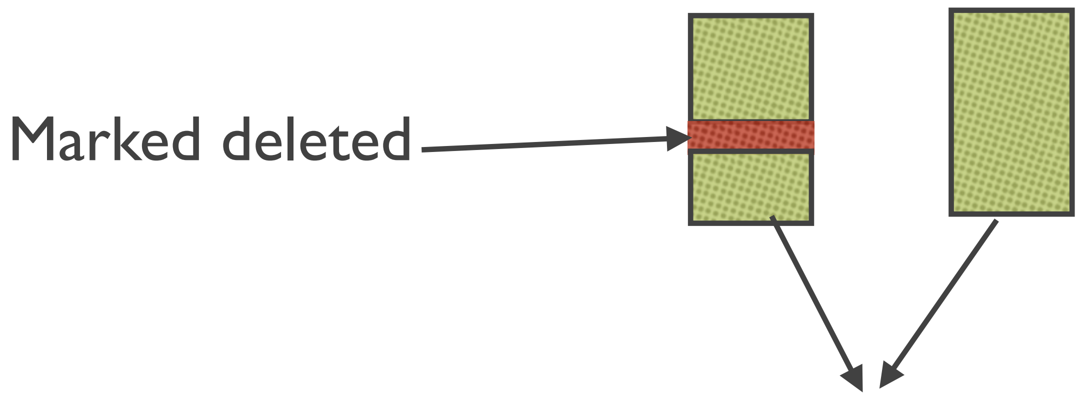


Lucene Segment Merging (Insert only)

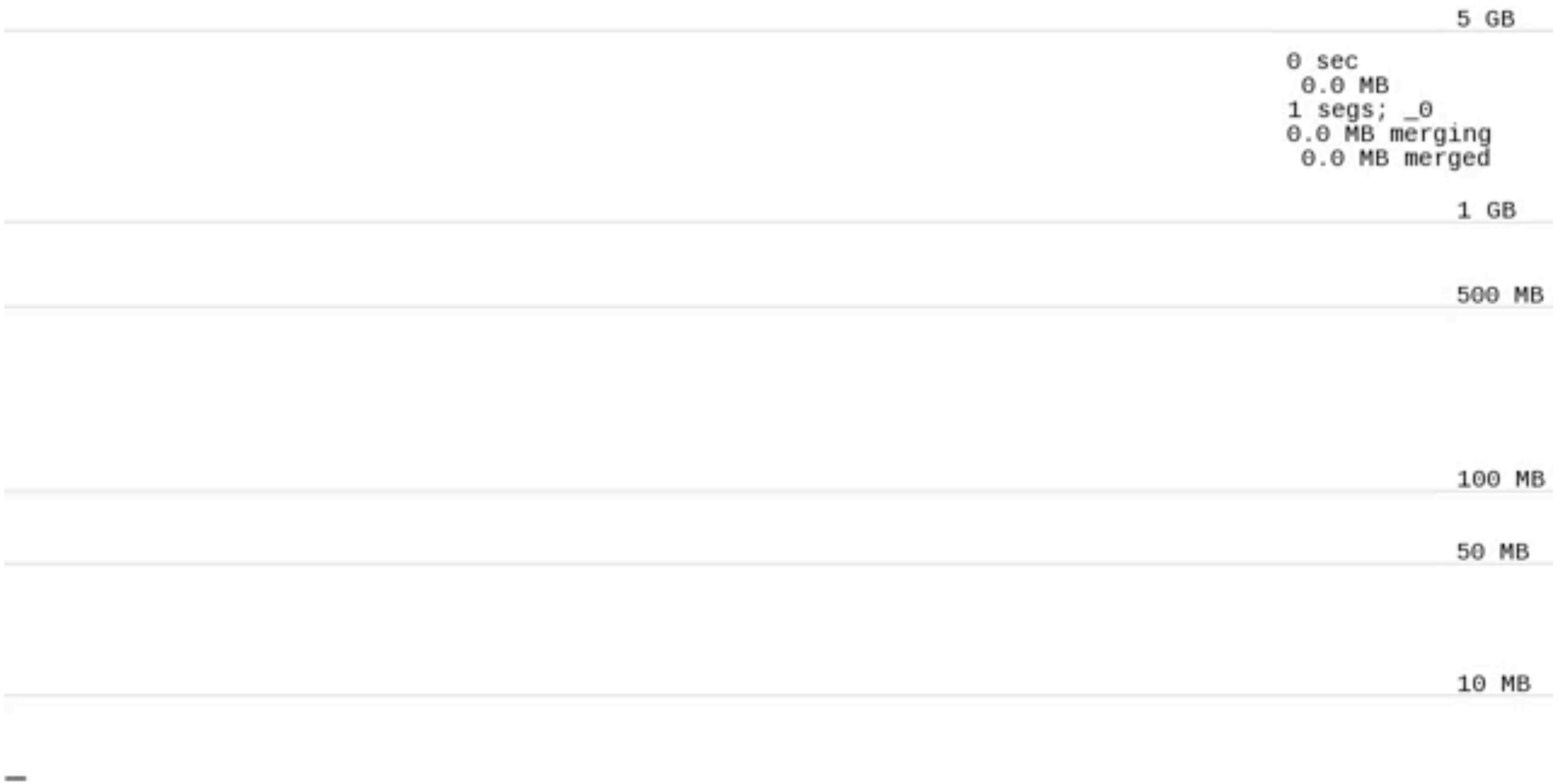


Lucene Dynamic Indexing

- How do deletes work?
- When documents are deleted
  - They are marked deleted in the segments
- When are they purged?



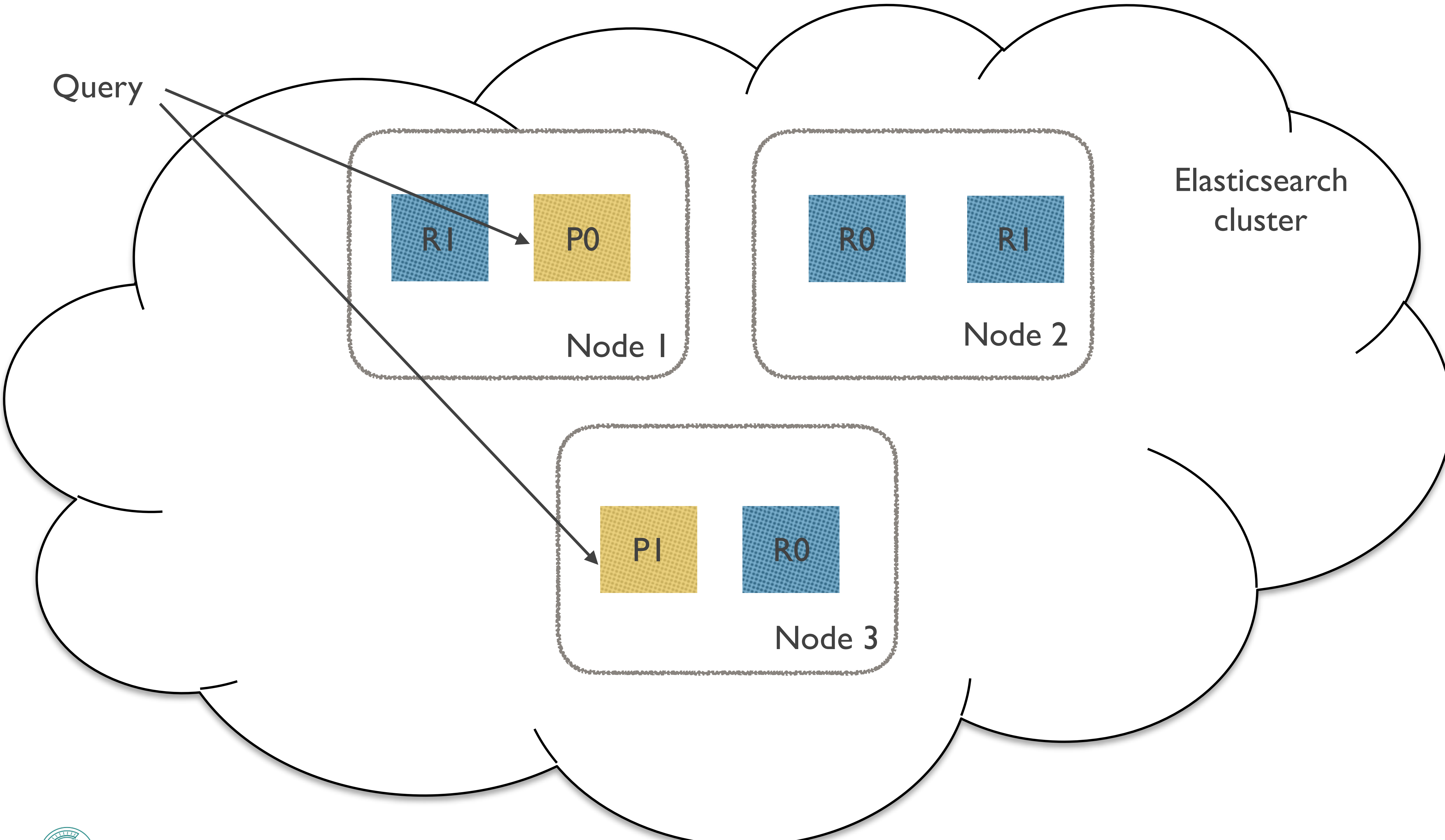
Lucene Segment Merging with Deletions



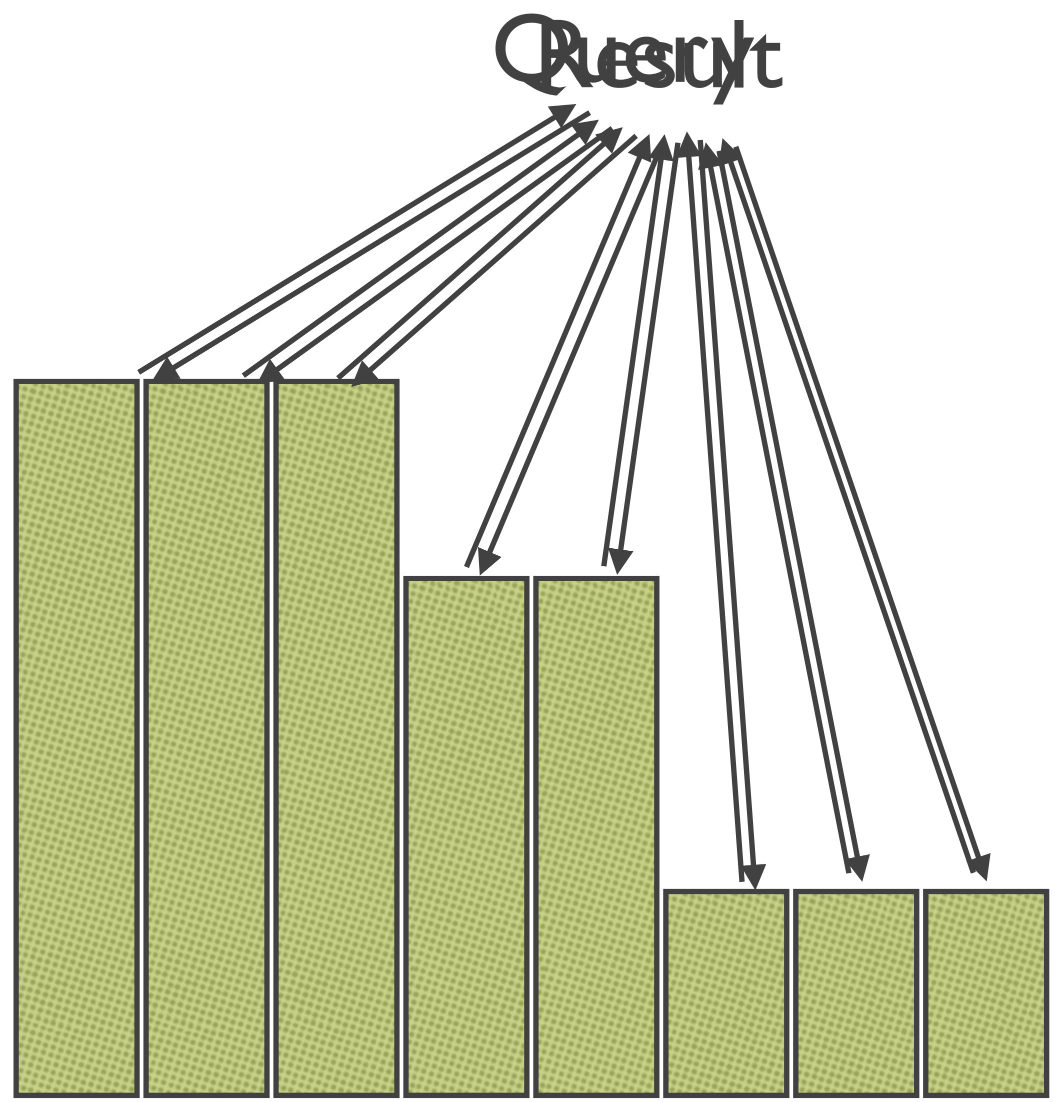
max planck institut
informatik

Source: http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html

Query Processing in Elasticsearch



Query Processing



Segments

Outline

3.1. Motivation

3.2. Index Construction & Maintenance

3.3. Static Index Pruning

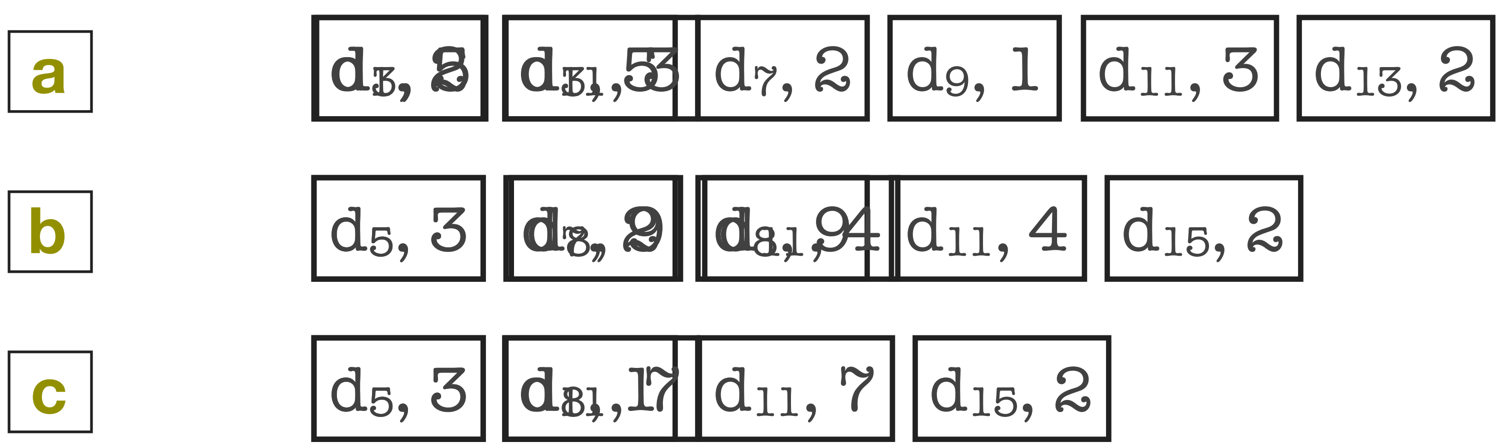
3.4. Document Reordering

3.5. Query Processing

max planck institut
informatik

3.3. Static Index Pruning

- **Static index pruning** is a form of **lossy compression** that
  - **removes postings** from the inverted index
  - allows for **control of index size** to make it fit, for instance, into main memory or on low-capacity device (e.g., smartphone)



- **Dynamic index pruning**, in contrast, refers to query processing methods (e.g., WAND or NRA) that avoid reading the entire index

max planck institut informatik

Term-Centric Index Pruning

- Carmel et al. [3] propose **term-centric** static index pruning
- Idea: Remove postings from posting list for term v that are **unlikely to contribute** to top-k result of query including v
- Algorithm: For each term v
  - **determine k-th highest score** z_v of any posting in posting list for v
  - **remove** all postings having a score less than $\varepsilon \cdot z_v$
- Despite its simplicity the method guarantees for any query q consisting of $|q| < 1 / \varepsilon$ terms a "close enough" top-k result

max planck institut informatik

MAX-PLANCK-GESELLSCHAFT

Document-Centric Index Pruning

- Büttcher and Clarke [2] propose **document-centric** index pruning
- Idea: Remove postings for document d corresponding to **non-important terms** for which it is unlikely to be in the query result
- Importance of term v for document d is measured using its **contribution to the KL divergence** from background model D

$$P\left[\, v \mid \theta_d \,\right] \log \left(\frac{P\left[\, v \mid \theta_d \,\right]}{P\left[\, v \mid \theta_D \,\right]} \right)$$

- **DCP$_{Const}$** selects constant number k of postings per document
- **DCP$_{Rel}$** selects a percentage λ of postings per document

max planck institut informatik

Term-Centric vs. Document-Centric

- Büttcher and Clarke [3] compare term-centric (TCP) and document-centric (DCP) index pruning on TREC Terabyte
  - **Okapi BM25** as baseline retrieval model
  - **on-disk inverted index**: 12.9 GBytes, 190 ms response time
  - **pruned in-memory inverted index**: 1 GByte, 18 ms response time

[**TREC 2004 Terabyte queries (topics 701-750)**]

	BM25 Baseline	$DCP_{Rel}^{(\lambda=0.062)}$	$DCP_{Const}^{(k=21)}$	$TCP_{(n=16000)}^{(k=24500)}$
P@5	0.5224	0.5020	0.4735	0.4490*
P@10	0.5347	0.4837	0.4755	0.4347*
P@20	0.4959	0.4490	0.4224	0.4163
MAP	0.2575	0.1963	0.1621**	0.1808

[**TREC 2005 Terabyte queries (topics 751-800)**]

	BM25 Baseline	$DCP_{Rel}^{(\lambda=0.062)}$	$DCP_{Const}^{(k=21)}$	$TCP_{(n=16000)}^{(k=24500)}$
P@5	0.6840	0.6760	0.6000**	0.5640**
P@10	0.6400	0.5980	0.5300*	0.5380**
P@20	0.5660	0.5310	0.4560**	0.4630**
MAP	0.3346	0.2465	0.1923**	0.2364

max planck institut informatik

Outline

3.1. Motivation

3.2. Index Construction & Maintenance

3.3. Static Index Pruning

3.4. Document Reordering

3.5. Query Processing

Index Compression

- **Sequences of non-decreasing integers** (here: document identifiers) in posting lists are compressed using
  - **delta encoding** representing elements as **difference to predecessor**

$$\langle 1, 7, 11, 21, 42, 66 \rangle \dashrightarrow \langle 1, 6, 4, 10, 21, 24 \rangle$$

  - **Variable-byte encoding: (aka. 7-bit encoding) represents integers (e.g., deltas of term offsets) as sequences of 1 continuation + 7 data bits**

docIDs	624	629	914
gaps	0	5	285
VB Code	00000100 11110000	10000101	00000100 10011101

  - **Gamma encoding:** unary code to represent length followed by offset binary of an integer but with leading 1 removed
    - e.g. 13 = 1101 = 1110101

3.4 Document Reordering

- **Document reordering methods** seek to **improve compression** effectiveness by **assigning document identifiers** so as to obtain **small gaps**
- **Content based document reordering**
- **K-means clustering**
  - similar documents get closer document ids
- **K-Scan**
  - Single scan k-means
- **URL-based** document id assignment

max planck institut informatik

Content-Based Document Reordering

- Silvestri et al. [7] develop methods for the scenario when **only document contents** are available but no meta-data (e.g., URL)
- Intuition: **Similar documents**, having many terms in common, should be assigned **numerically close document identifiers**
- **Documents** are modeled as **sets** (not bags) **of terms**
- **Document similarity** is measured using the **Jaccard coefficient**

$$J(d_i, d_j) = \frac{|d_i \cap d_j|}{|d_i \cup d_j|}$$

max planck institut informatik

Top-Down Bisecting

- Algorithm: TDAssign(document collection D)

  // split D into equal-sized partitions D_L and D_R

  pick representatives d_L and d_R (e.g., randomly)

  if $(|D_L| \geq |D| / 2) \vee (|D_R| \geq |D| / 2)$

      assign d to smaller partition

  else if $J(d, d_L) > J(d, d_R)$

      assign d to D_L

  else

      assign d to D_R

  return TDAssign(D_L) $\oplus$ TDAssign(D_R)

- TDAssign has **time complexity** in $O(|D| \log |D|)$

kScan

- Algorithm: kScan(document collection D)

  // split D into k equal-sized partitions D_i

  n = |D|

  for i = 1 … k

  d_i = longest document from D

  assign n/k documents with highest similarity $J(d, d_i)$ to D_i

  $D = D \setminus D_i$

  return < d from D_1> $\oplus$ … $\oplus$ <d from D_k>

- kScan has **time complexity** in $O(k\ |D|)$

- kScan **outperforms** TDAssign **in terms of compression effectiveness** (bits per posting) in experiments on collections of web documents

max planck institut
informatik

URL-Based Document Reordering

- Silvestri [8] examines the effectiveness of URL-based document reordering when compressing **collections of web documents**

- Intuition: Documents with **lexicographically close URLs** tend to have **similar contents** (e.g., www.x.com/a and www.x.com/b)

- Algorithm:
  - **sort** documents **lexicographically according to their URL**
  - **assign** consecutive document identifiers (1 … |D|)

max planck institut informatik

Content-Based vs. URL-Based

- Silvestri [8] reports experiments conducted on a **large-scale crawl of the Brazilian Web** (about 6 million documents)

	VByte	Gamma	Delta
Random	11.40	12.72	12.71
URL	**9.72**	**7.72**	**7.69**
kScan	9.81	8.82	8.80

- URL-based document ordering **outperforms** content-based document ordering (kScan), requiring **fewer bits per posting on average**

Outline

3.1. Motivation

3.2. Index Construction & Maintenance

3.3. Static Index Pruning

3.4. Document Reordering

3.5. Query Processing

Query Processing

- Query processing methods operate on inverted index
  - holistic query processing methods determine the full query results
    (e.g., document-at-a-time and term-at-a-time)
  - top-k query processing methods (aka. dynamic index pruning) determine only the top-k query result and avoid reading posting lists completely
    - Fagin's TA and NRA for score-ordered posting lists
    - WAND and Block-Max WAND for document-ordered posting lists

WAND

- Broder et al. [1] describe **WAND** (weak AND) as a **top-k query processing method** for document-ordered posting lists
  - DAAT-style traversal of posting lists in parallel
  - assumes that the **maximum score** max(i) per posting list **is known**
  - **pivoted cursor movement** based on current top-k result
    - let min_k denote the worst score in the current top-k result **(1)**
    - sort cursors for posting lists based on their current document identifier cdid(i) **(2)**
    - pivot document identifier p is the smallest cdid(j) such that **(3)**

$$\mathrm{min}_\mathrm{k} < \sum_{i \leq j} \max(i)$$

    - move all cursors i with cdid(i) < p up to pivot p

WAND

- Example: Pivoted cursor movement based on top-1 result

Top-1
d_1 : 8

a	d_1, 2	…	d_3, 1	…	max(a) = 3
b	d_1, 3	…	d_2, 3	…	max(b) = 3
c	d_1, 3	…	d_9, 3	…	max(c) = 3

	cdid	Σ	
	d_2, 3	3	
min_k = 8	d_3, 1	6	p = d_9
	d_9, 3	**9**	
(1)	(2)		(3)

- It is safe to move the cursor for posting lists a and b forward to d_9

Block-Max WAND

- Ding and Suel [4] propose the **block-max inverted index**
  - store posting list as sequence of **compressed posting blocks**
  - each block contains a **fixed number of postings** (e.g., 64)
  - keep **minimum document identifier** and **maximum score** per block

	(1, 5)		(7, 2)		(11, 3)	
a	d_1, 2	d_3, 5	d_7, 2	d_9, 1	d_{11}, 3	d_{13}, 2

max(a) = 5

these are available without having to decompress the block

Block-Max WAND

- Pivoted cursor movement considering per-block maximum scores
  - determine **pivot** p according to WAND
  - perform **shallow cursor movement** for all cursors i with cdid(i) < p
    (i.e., do **not decompress** if a new posting block is reached)
  - **if** any document from current blocks can make it into top-k, i.e.:

$$\mathrm{min_k} < \sum_{i:cdid(i) \leq p} \text{block\_max}(i)$$

    perform **deep cursor movement** (i.e., decompress posting blocks) and continue as in WAND
  - **else** move cursor with minimal **cdid(i)** to

$$\min\left(\min_{i:cdid(i)\leq p} \text{next\_block\_mdid}(\mathrm{i}), cdid(p+1)\right)$$

max planck institut
informatik

Block-Max WAND

- Example: Pivoted cursor movement based on top-1 result

Top-1
$d_1 : 8$

a	$d_1, 2$	…	$d_3, 1$ (2, 1)	…	(5, 1) shallow	(11, 3)		max(a) = 3
b	$d_1, 3$	…	$d_2, 3$ (2, 3)	…	(4, 1) shallow	(10, 2)		max(b) = 3
c	$d_1, 3$	…	$d_9, 3$ (7, 3)	…	(14, 1)	(17, 2)		max(c) = 3
d	$d_2, 3$	…	$d_{11}, 3$	…				max(d) = 3

Summary

- **Inverted indexes** can be **efficiently constructed** offline by using external memory sort or MapReduce
- **Inverted indexes** can be **efficiently maintained** by using logarithmic/geometric partitioning
- Index maintenance and query processing in **elasticsearch**
- **Static index pruning methods** reduce index size by systematically **removing postings**
- **Document reordering methods** reduce index size by assigning document identifiers so as to yield **smaller gaps**
- **Query processing** on document-ordered inverted indexes can be greatly sped up by **pivoted cursor movement** as part of WAND and Block-Max WAND

References

[1] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, J. Zien: *Efficient Query Evaluation using a Two-Level Retrieval Process*, CIKM 2003

[2] S. Büttcher and C. L. A. Clarke: *A Document-Centric Approach to Static Index Pruning in Text Retrieval Systems*, CIKM 2006

[3] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, Y. S. Maarek, A. Soffer: *Static Index Pruning for Information Retrieval Systems*, SIGIR 2001

[4] S. Ding and T. Suel: *Faster Top-k Retrieval using Block-Max Indexes*, SIGIR 2011

[5] N. Leser, A. Moffat, J. Zobel: *Efficient Online Index Construction for Text Databases* ACM TODS 33(3), 2008

[6] N. Lester, J. Zobel, H. Williams: *Efficient Online Index Maintenance for Inverted Lists*, IP&M 42, 2006

[7] F. Silvestri, S. Orlando, R. Perego: *Assigning Identifiers to Documents to Enhance the Clustering Property of Fulltext Indexes*, SIGIR 2004

References

[8] **F. Silvestri:** *Sorting Out the Document Identifier Assignment Problem*, ECIR 2007

[9] **L. Wu, W. Lin, X. Xiao, Y. Xu:** *LSII: An Indexing Structure for Exact Real-Time Search on Microblogs*, ICDE 2013

For more on index compression refer to the slides from IRDM 2015 http://resources.mpi-inf.mpg.de/departments/d5/teaching/ws15_16/irdm/slides/irdm2015-ch11-handout.pdf

For query processing like top-k NRA and TA algorithms refer to http://resources.mpi-inf.mpg.de/departments/d5/teaching/ws15_16/irdm/slides/irdm2015-ch12-queryprocessing.pdf

Additionally you can also refer to Chapter 5 in Introduction to Information retrieval by Christopher D. Manning et.al.

- Some slides were borrowed from Prof. Klaus Berberich