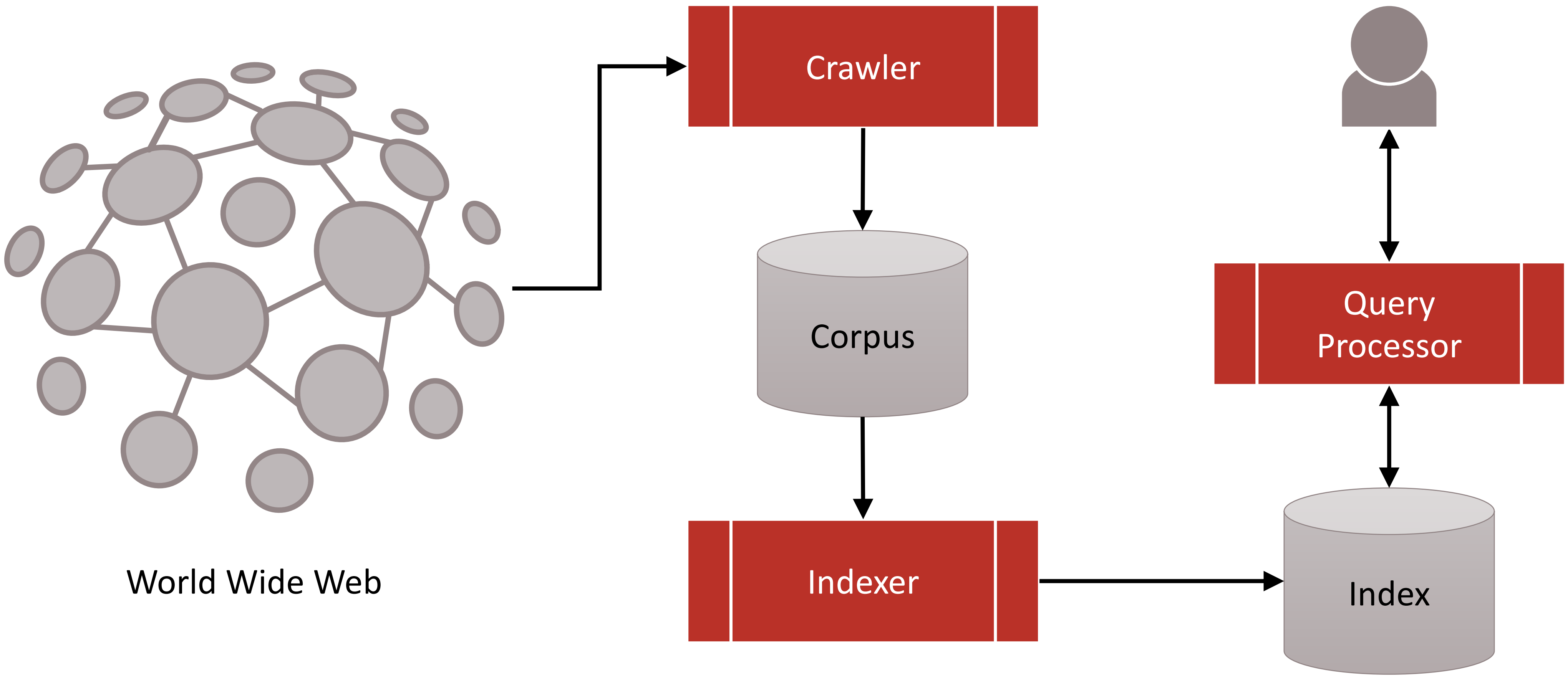
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Information Retrieval

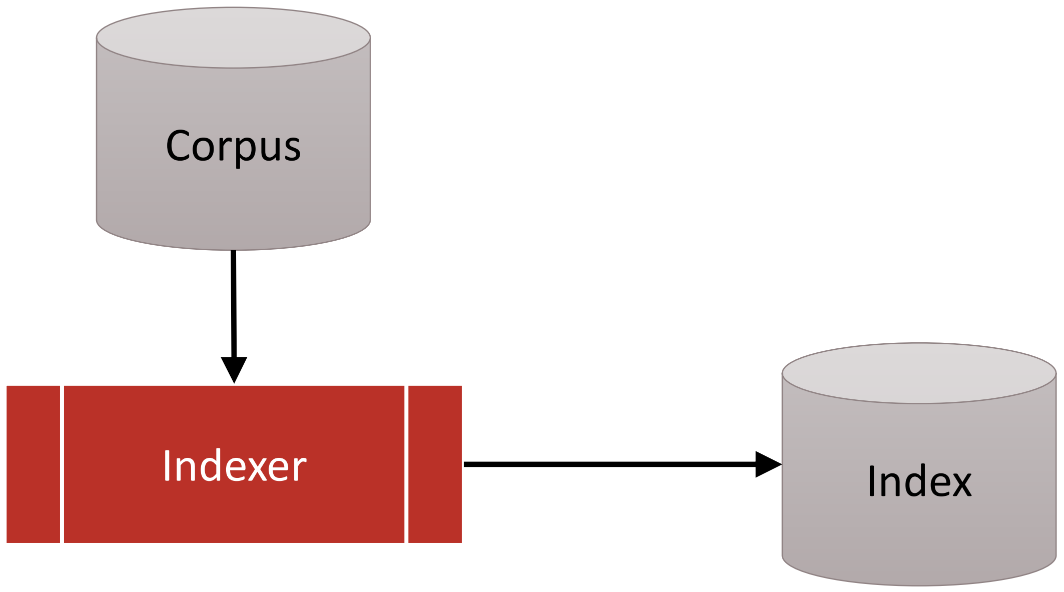
Document Indexing

Rodrygo L. T. Santos
rodrygo@dcc.ufmg.br

Search components



Search components



Key challenges

Support effective retrieval

- Extract meaningful document features
- Both topical and quality features

Support efficient retrieval

- Quick scoring of matched documents

Success metrics

Quality metrics

- Content utility: fraction of spam, exact/near dups
- Effectiveness: measured at ranking time

Performance metrics

- Compactness: size of the index in bytes
- Deployment cost: time to build / update index

Document prefiltering

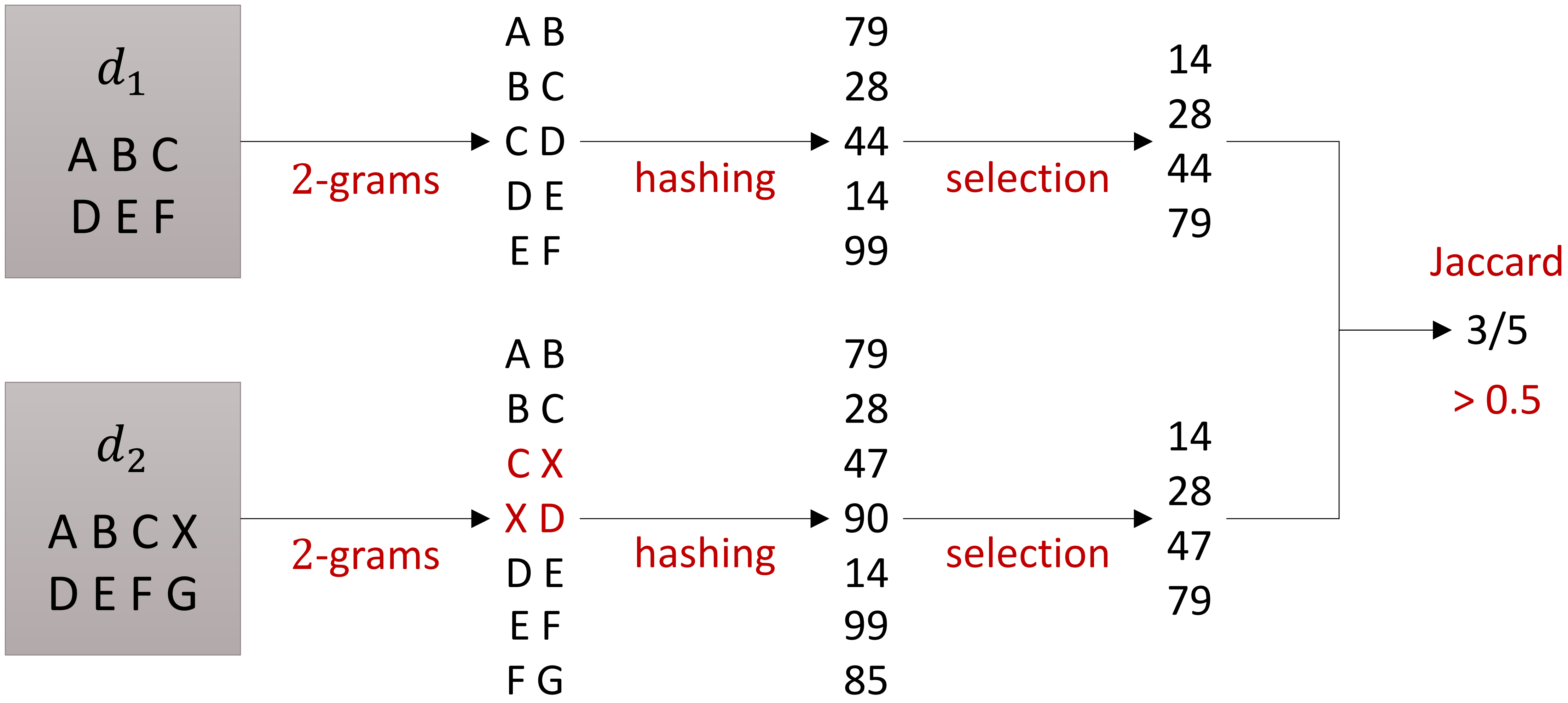
Detecting spam documents

- Term spamming → text classification
- Link spamming → trust propagation

Detecting documents with duplicate content

- Exact duplicates → compare hash values
- Near duplicates → compare *shingles* instead

Near duplicates via n-shingling



Document features

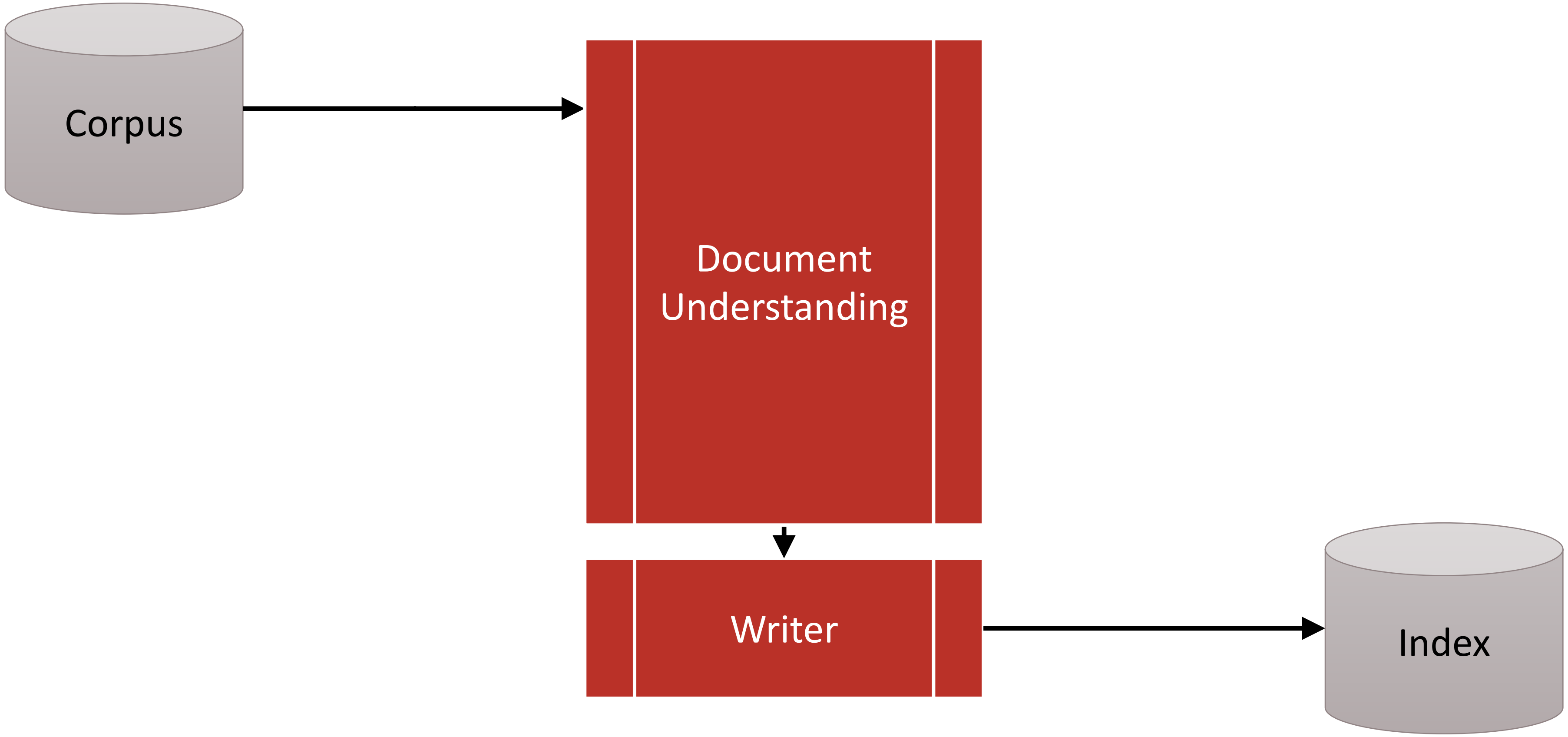
Features computed offline

- Content: spam score, domain quality score
- Web graph: PageRank, HostRank
- Usage: click count, CTR, dwell time

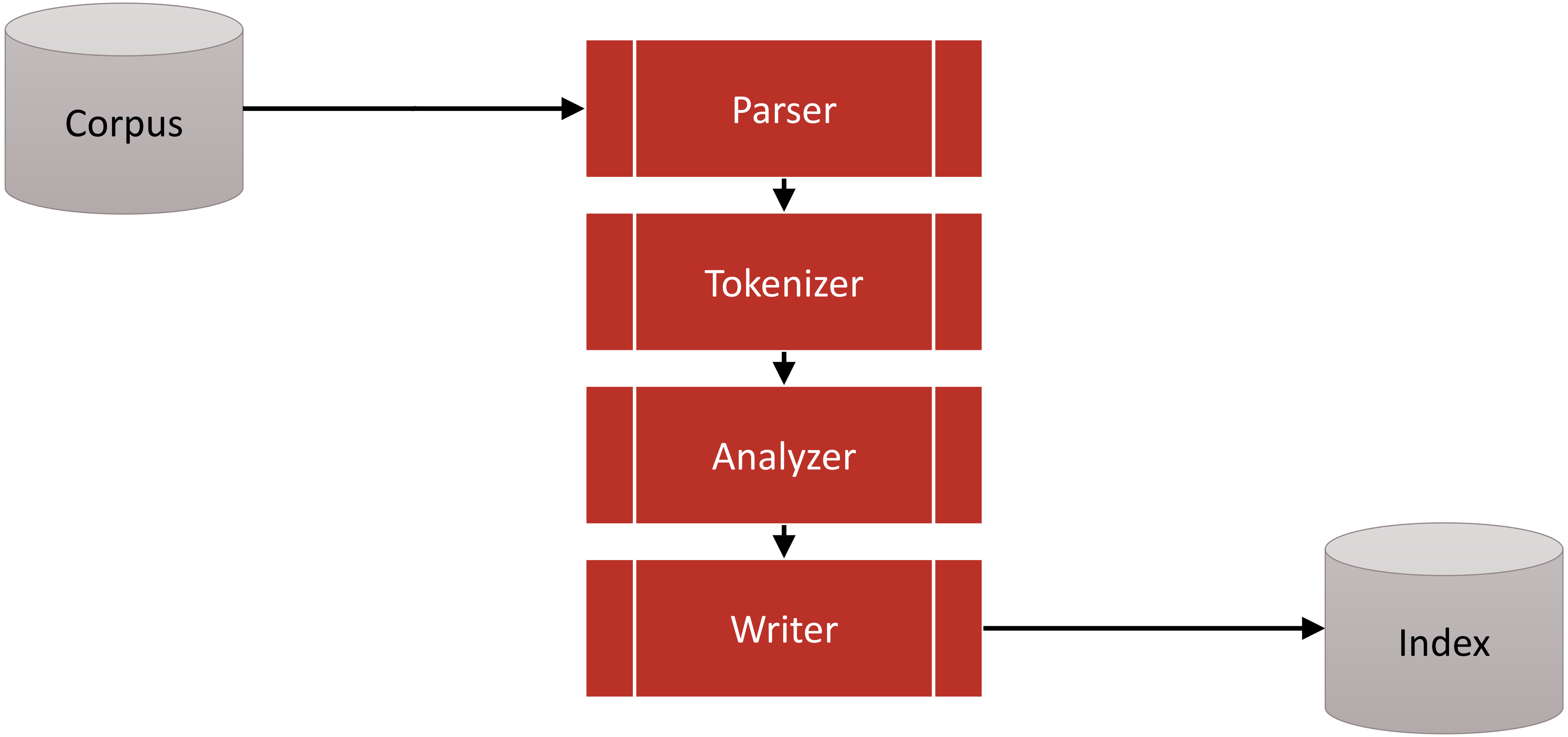
Features computed online

- Query-document similarity: TF-IDF, BM25, etc.

Indexing overview



Indexing overview



Document understanding

Parsing + tokenization

- Turn raw text into indexing terms

Token analysis

- Discriminative power
- Equivalence classing
- Phrasing, scoping

Document indexing

Indexing makes crawled documents searchable

- Efficient searching requires appropriate structures

Abandoned indexing data structures

- Suffix arrays, signature files

Currently used data structure

- Inverted index

Example “corpus”

d_1	Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.
d_2	Fish keepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.
d_3	Tropical fish are popular aquarium fish, due to their often bright coloration.
d_4	In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

Incidence matrix

	d_1	d_2	d_3	d_4
and	■	□	□	□
aquarium	□	□	■	□
are	□	□	■	■
around	■	□	□	□
as	□	■	□	□
both	■	□	□	□
bright	□	□	■	□
coloration	□	□	■	■
derives	□	□	□	■
due	□	□	■	□
environments	■	□	□	□
fish	■	■	■	■

Inverted index: incidence

and	1			
aquarium	3			
are	3	4		
around	1			
as	2			
both	1			
bright	3			
coloration	3	4		
derives	4			
due	3			
environments	1			
fish	1	2	3	4

Inverted index: frequency

and	1:1			
aquarium	3:1			
are	3:1	4:1		
around	1:1			
as	2:1			
both	1:1			
bright	3:1			
coloration	3:1	4:1		
derives	4:1			
due	3:1			
environments	1:1			
fish	1:2	2:3	3:2	4:2

Inverted index: additional info

Term positions

- Exact phrases vs. proximity search

Document structure

- Field restrictions (e.g., date:, from:)
- Some fields more important (e.g., title, h1)

Inverted list compression

Key observation

- Standard representation wasteful

fish | 1:2 | 2:3 | 3:2 | 4:2

docid: 00000000000000000000000000000001 (32 bits)
frequency: 00000000000000000000000000000010 (32 bits)
$=$ 64 bits per posting
$\times$ 4 postings
$=$ 256 bits

Inverted list compression

Basic idea

- Lossless representation using fewer bits
- Several approaches [Catena et al., ECIR 2014]

Key benefits

- Reduced space and network / disk transfer costs

Cost: decompression overhead

Example: unary encoding

Consider encoding $k = 5_{10}$

$$5_{10} = 00000000000000000000000000000101_2 \text{ (32 bits)}$$

Unary encoding of k

- Write k 0's followed by one 1

$$5_{10} = 000001_2 \text{ (6 bits)}$$

Example: gamma encoding

Consider encoding $k = 5_{10}$

$$5_{10} = 00000000000000000000000000000101_2 \text{ (32 bits)}$$

Gamma encoding of k

- Let $l = \lfloor \log_2 k \rfloor$ (# binary digits of k minus 1)
- Let r be the binary repr. of k less most significant bit
- Write $unary(l)$ followed by r

$5_{10} = 00101_2$ (5 bits)

How about $k = 1{,}000{,}000$?

Uncompressed representation

$$1\mathrm{M}_{10} = 00000000000011110100001001000000_2$$ (32 bits)

Unary encoding

$$1\mathrm{M}_{10} = 00000000000000000000\ldots01_2$$ (1M+1 bits)

Gamma encoding

$$1\mathrm{M}_{10} = 000000000000000000011110100001001000000_2$$

(39 bits)

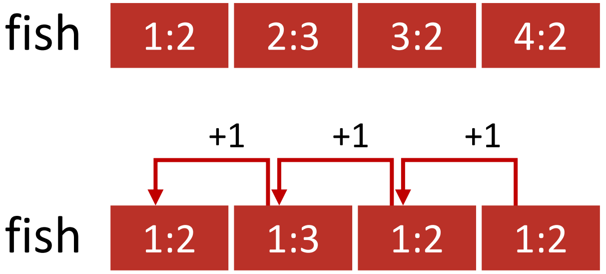
Inverted list compression

32 bit int cover -2,147,483,648 to +2,147,483,647

- Docids can get very large (~100B)

Large numbers require more bits

- Store delta gaps instead



Document identifier reordering

Goal: reassign document identifiers so that we obtain many small d-gaps, facilitating compression

old lists:	L1: 1 3 6 8 9	L2: 2 4 5 6 9	L3: 3 6 7 9
mapping:	1→1 2→9 3→2 4→7 5→8 6→3 7→5 8→6 9→4		
new lists:	L1: 1 2 3 4 6	L2: 3 4 7 8 9	L3: 2 3 4 5
old d-gaps:	2 3 2 1	2 1 1 3	3 1 2
new d-gaps:	1 1 1 2	1 3 1 1	1 1 1

Document identifier reordering

Key idea: assign similar documents close docids

- A term in one doc will likely be in similar docs

Clustering similar documents

- Assigns nearby IDs to documents in the same cluster

Sorting URLs alphabetically

- Pages from same site have high textual overlap

Index construction

```
function index(corpus)
  index = map()
  did = 0
  for document in corpus
    did += 1
    for (term, tf) in tokenize(document)
      if term not in index.keys()
         index[term] = list()
      index[term].append((did, tf))
  return index
```

Index construction

Equivalent to computing the transpose of a matrix

- Parse direct (document-term) occurrences
- Write out inverted (term-document) occurrences

Trivial in-memory implementation

- Does not scale even for moderately sized corpora
- On-disk processing typically needed

External indexing

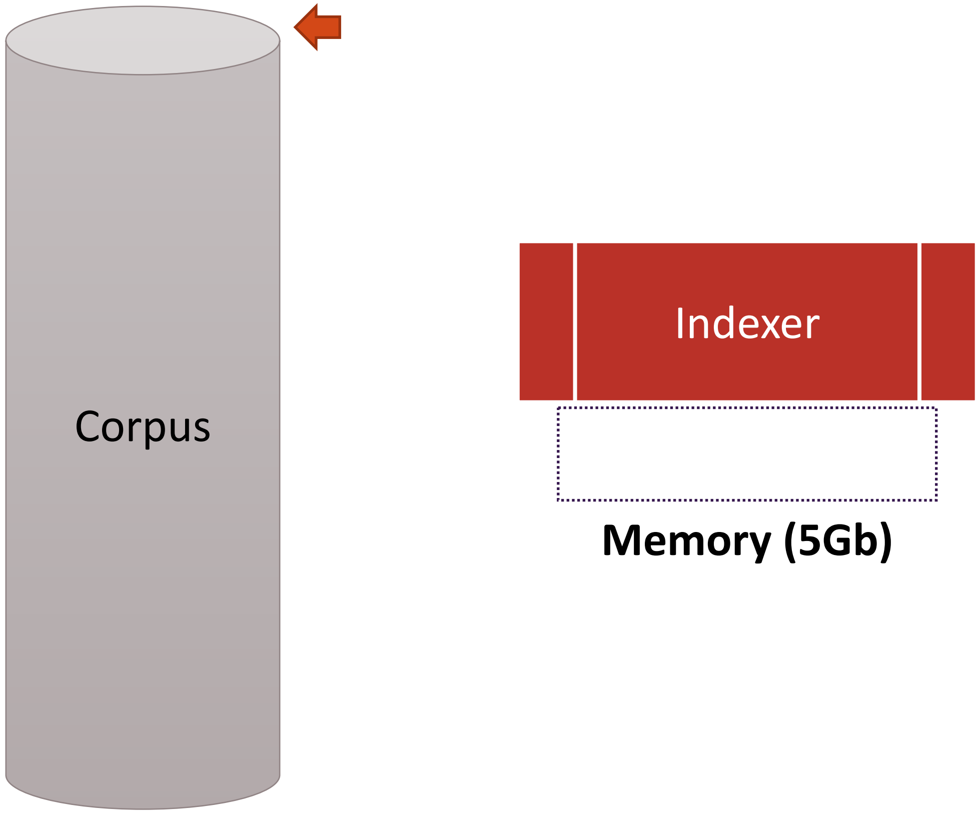
Merging addresses limited memory problem

- Build the inverted list structure until memory runs out
- Write partial index to disk, start making a new one

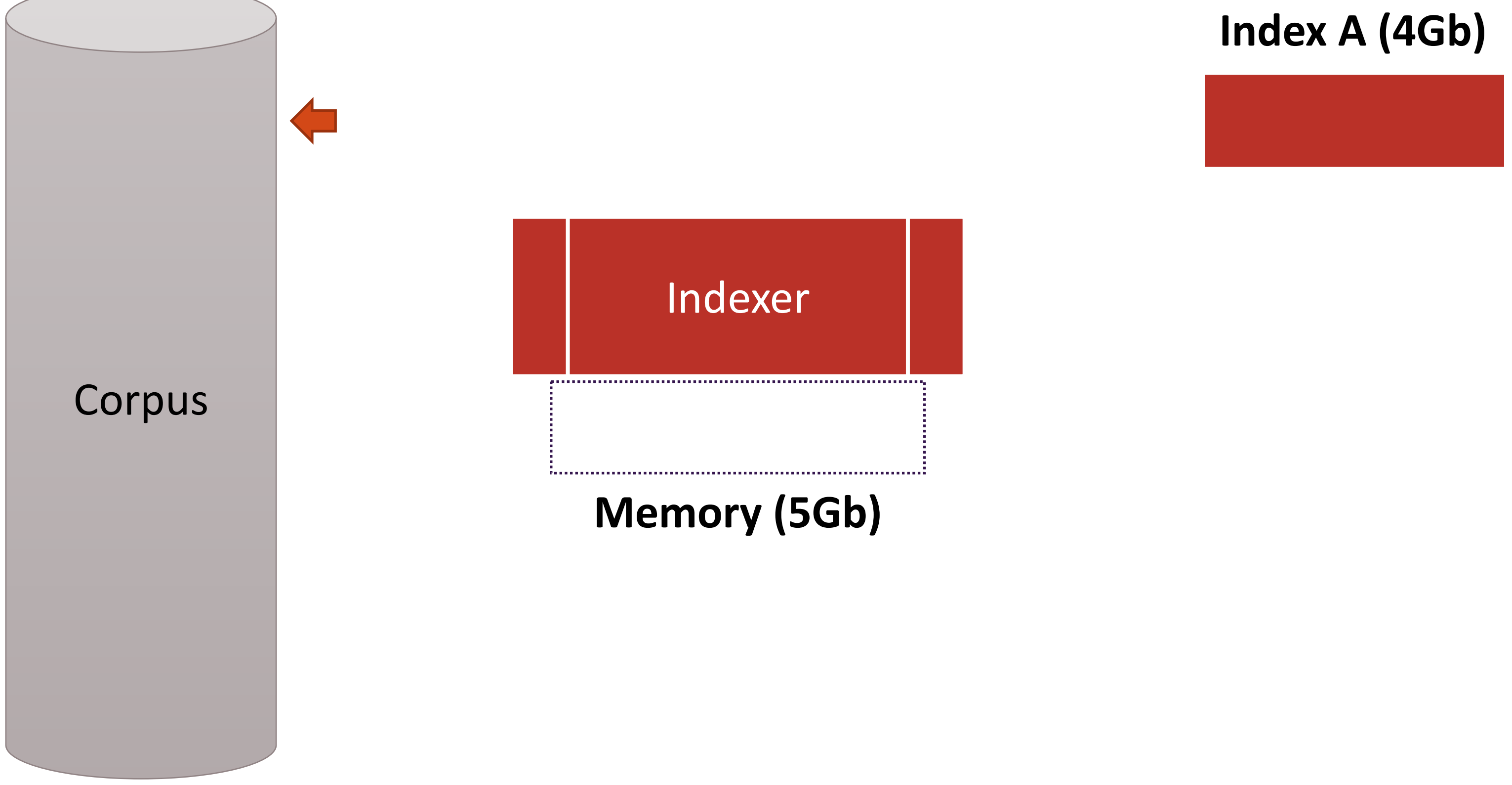
At the end of this process

- Merge many partial indexes in memory
- Equivalent to an external mergesort

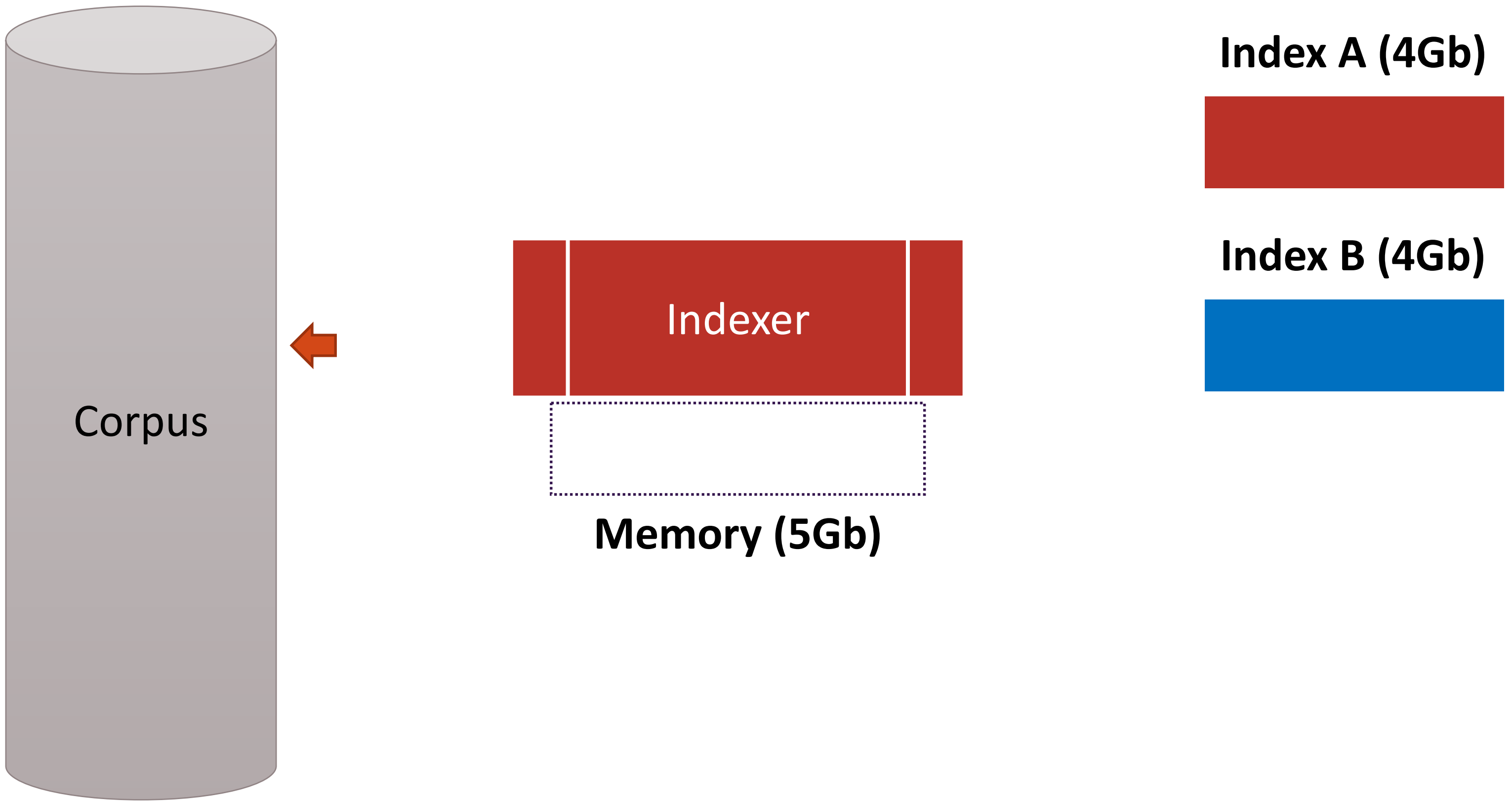
External indexing



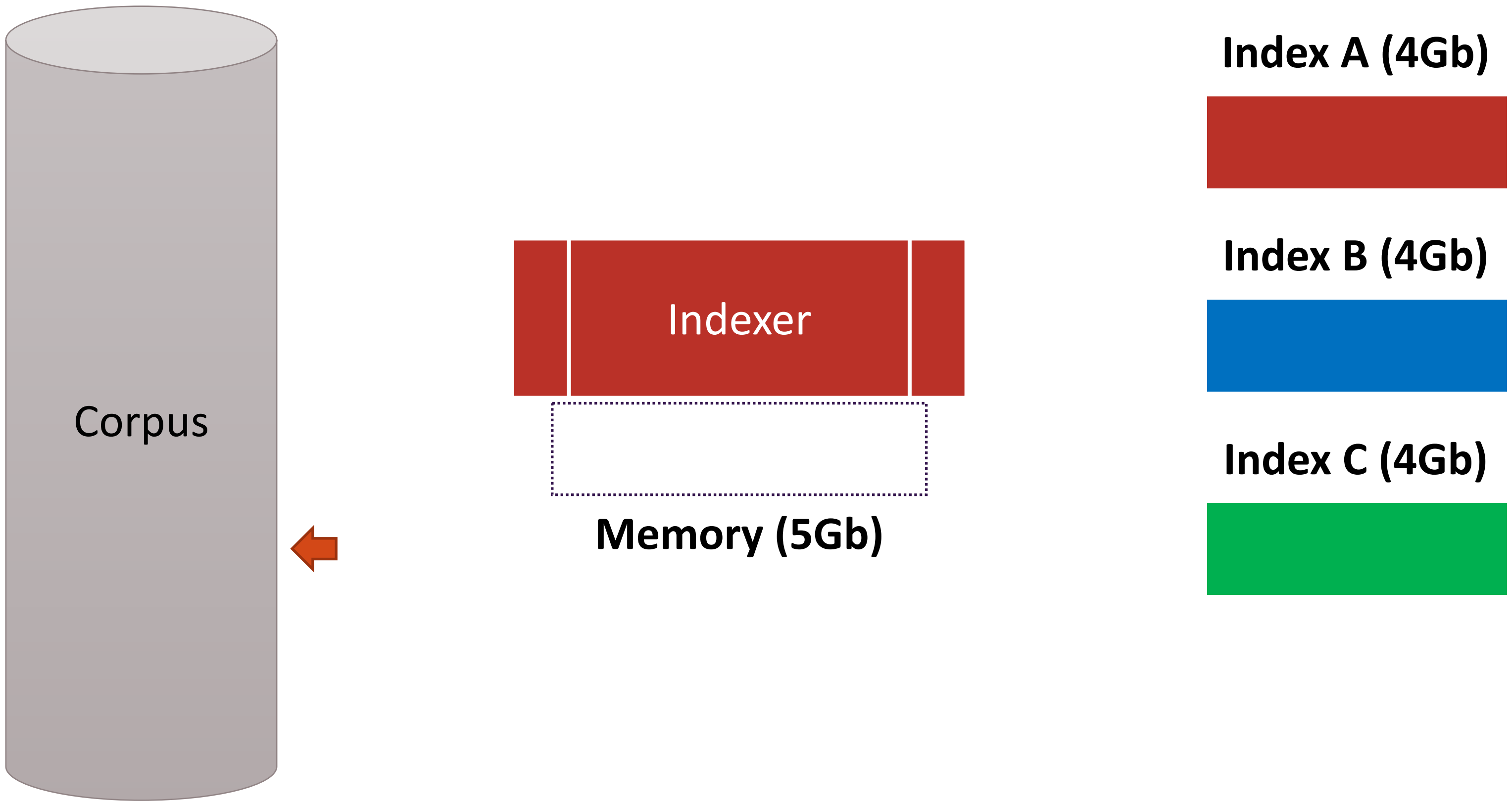
External indexing



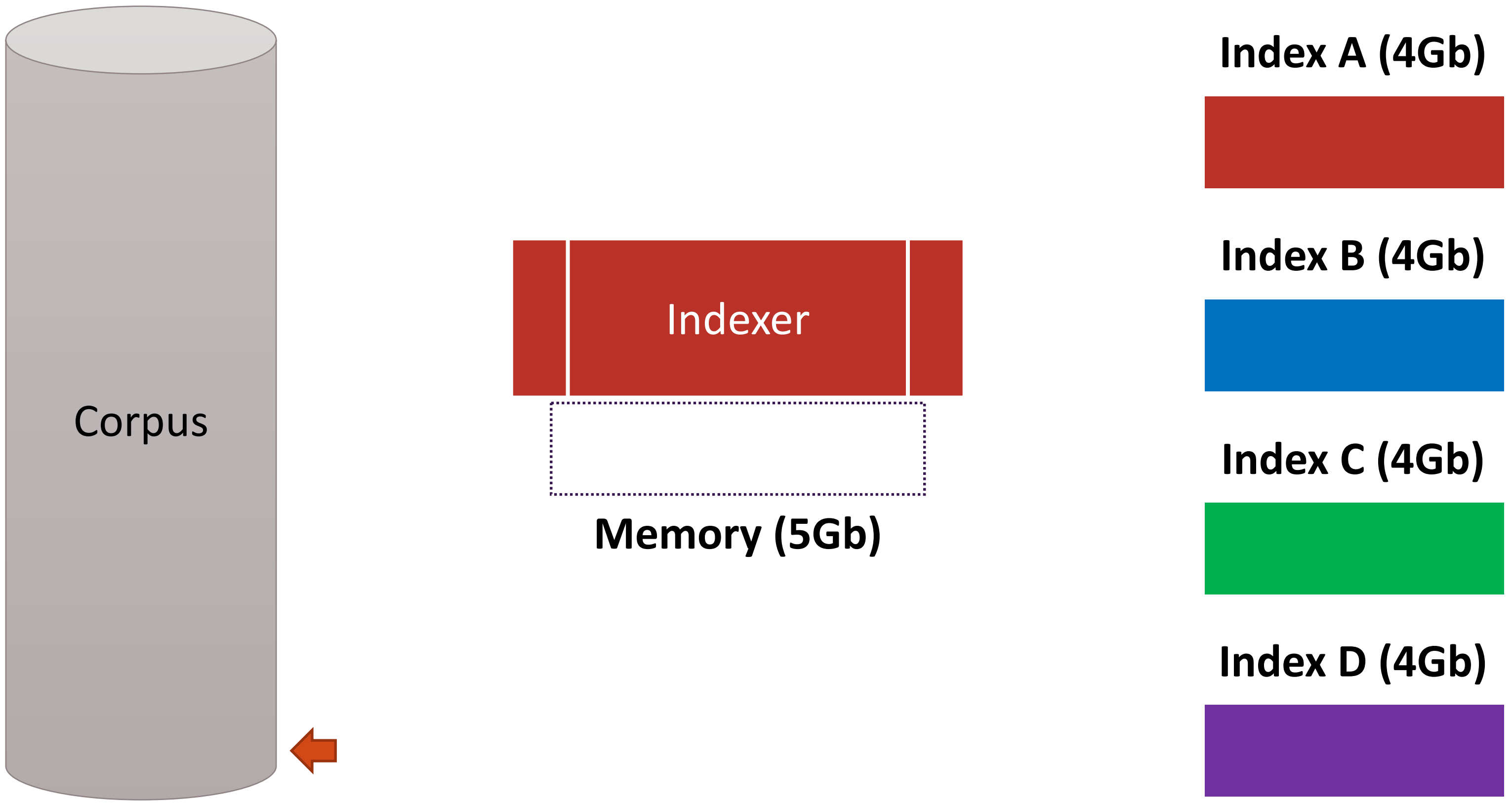
External indexing



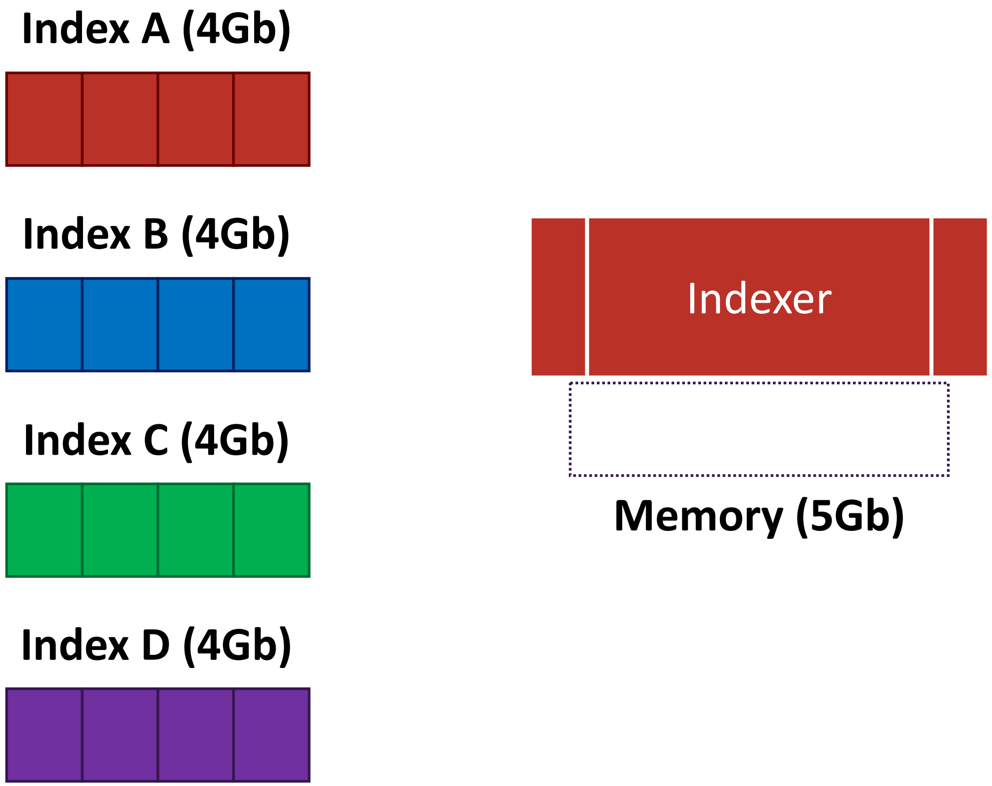
External indexing



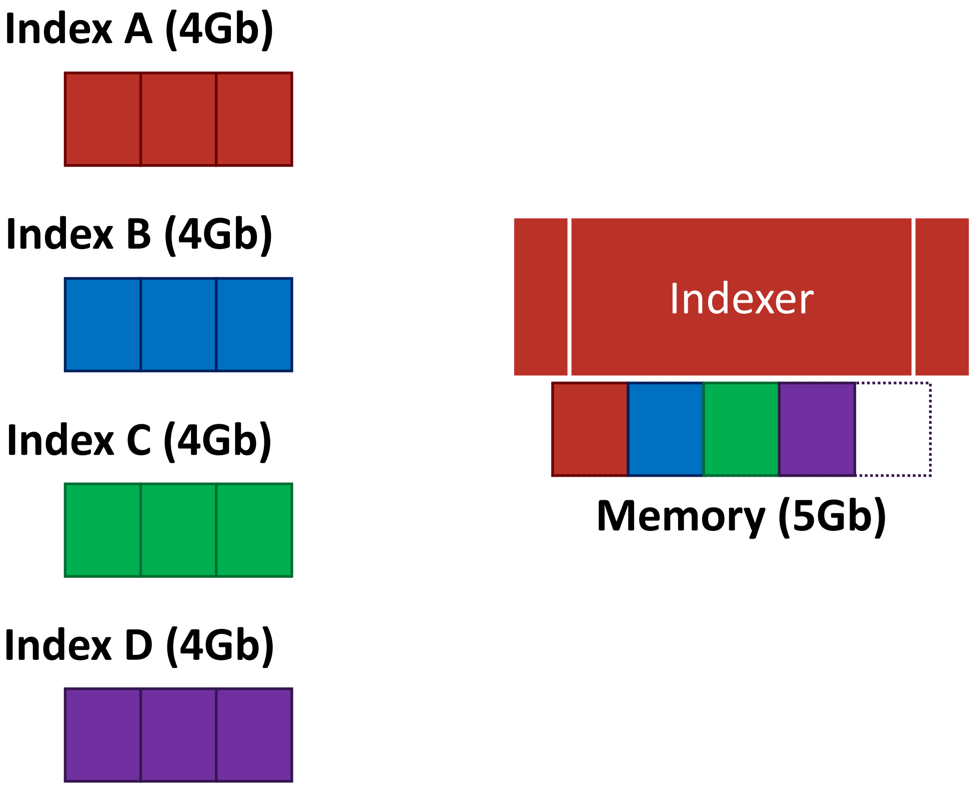
External indexing



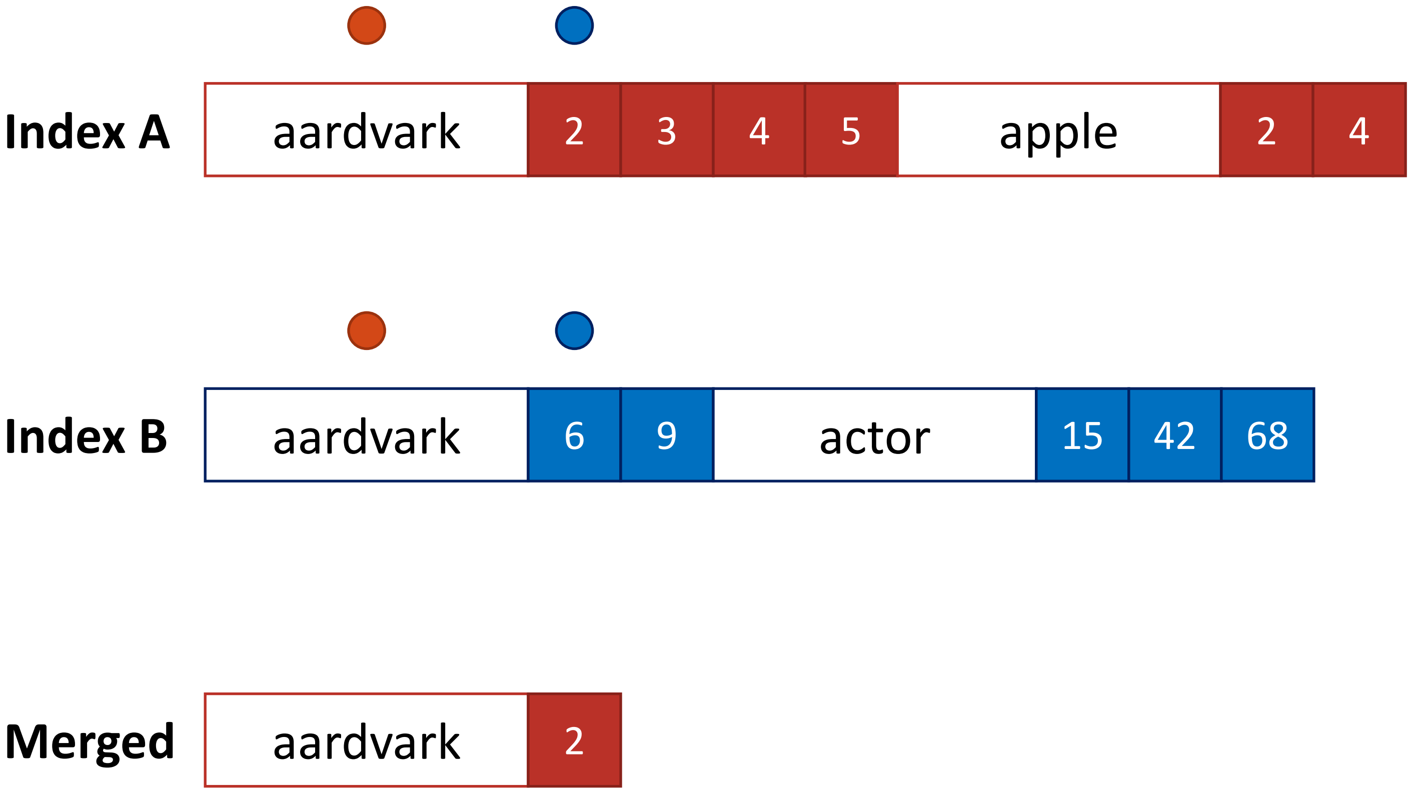
External merging



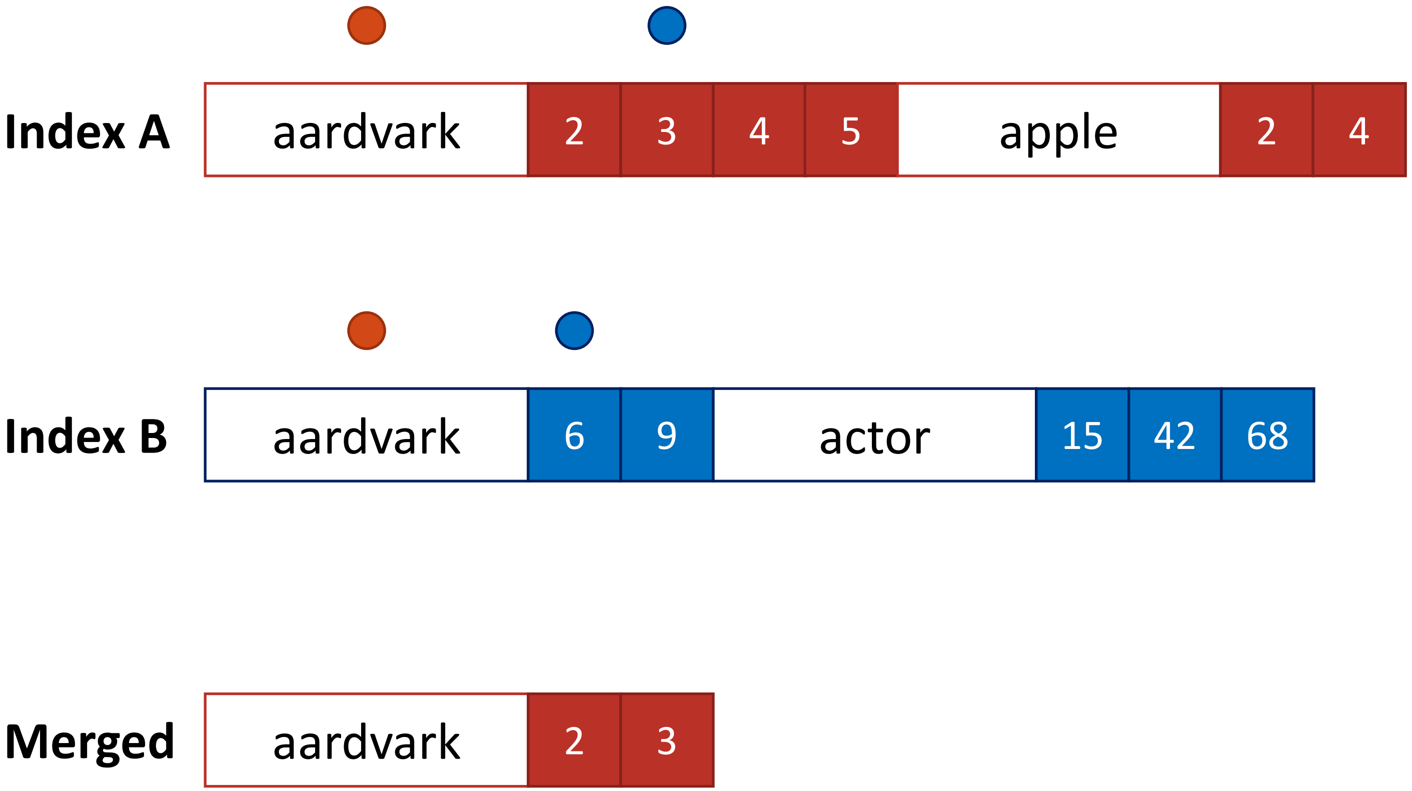
External merging



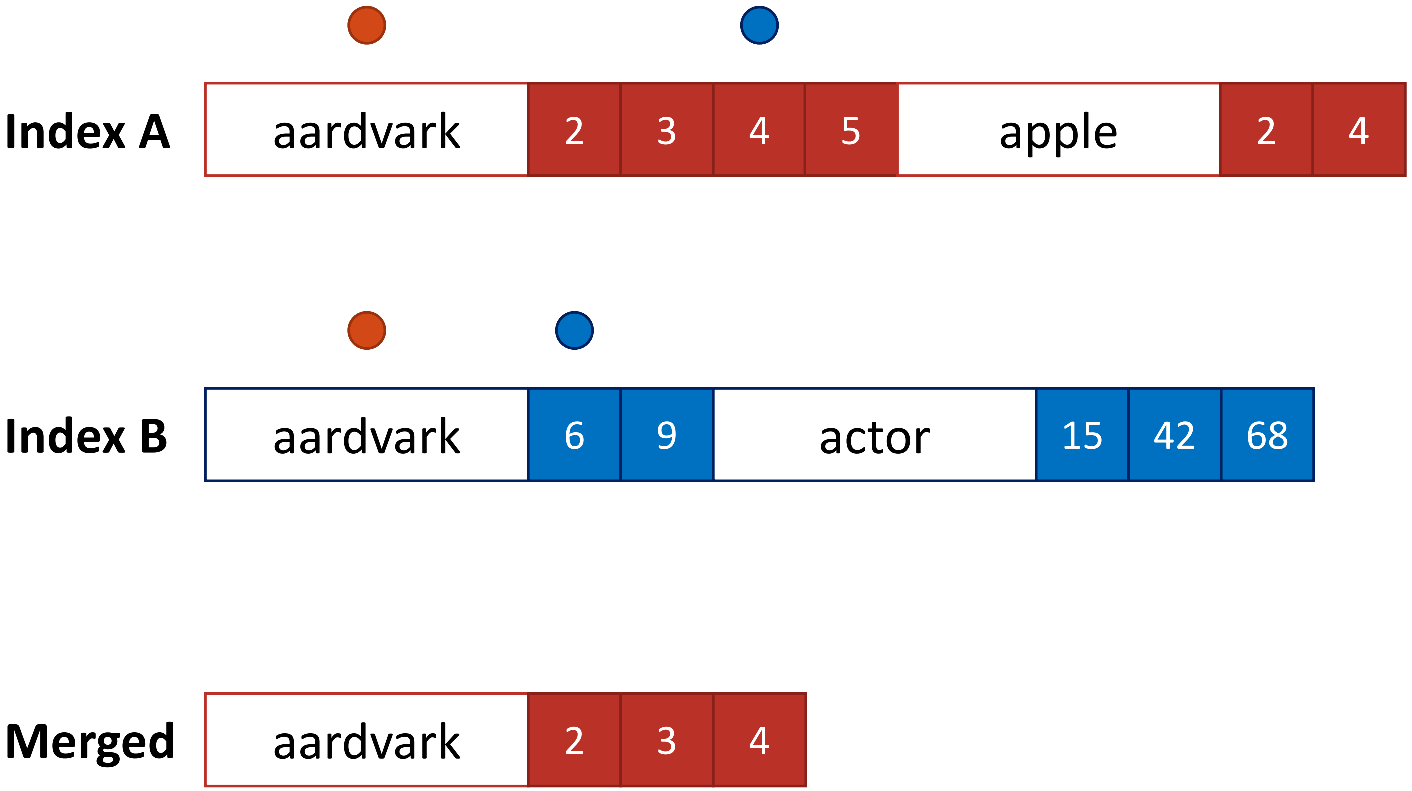
Internal merging



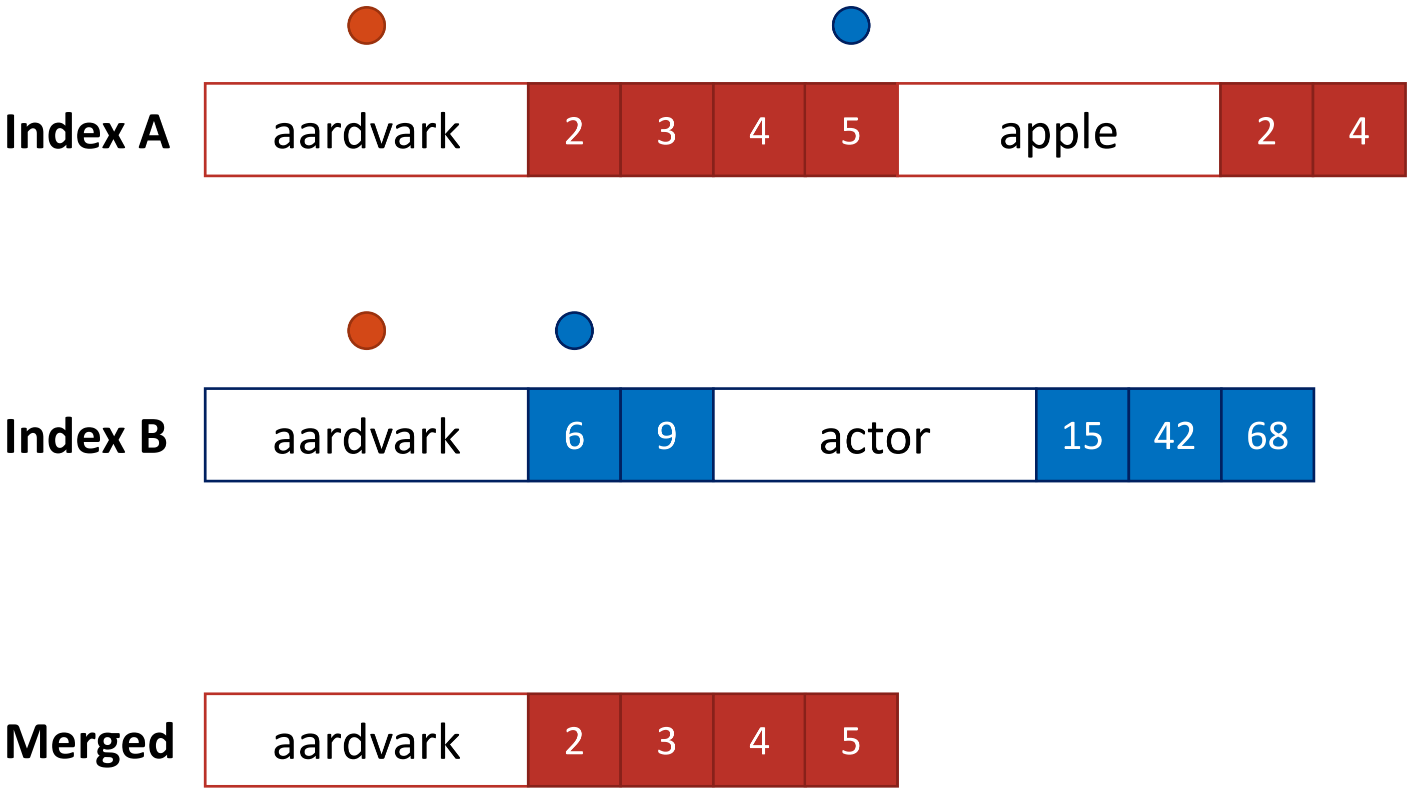
Internal merging



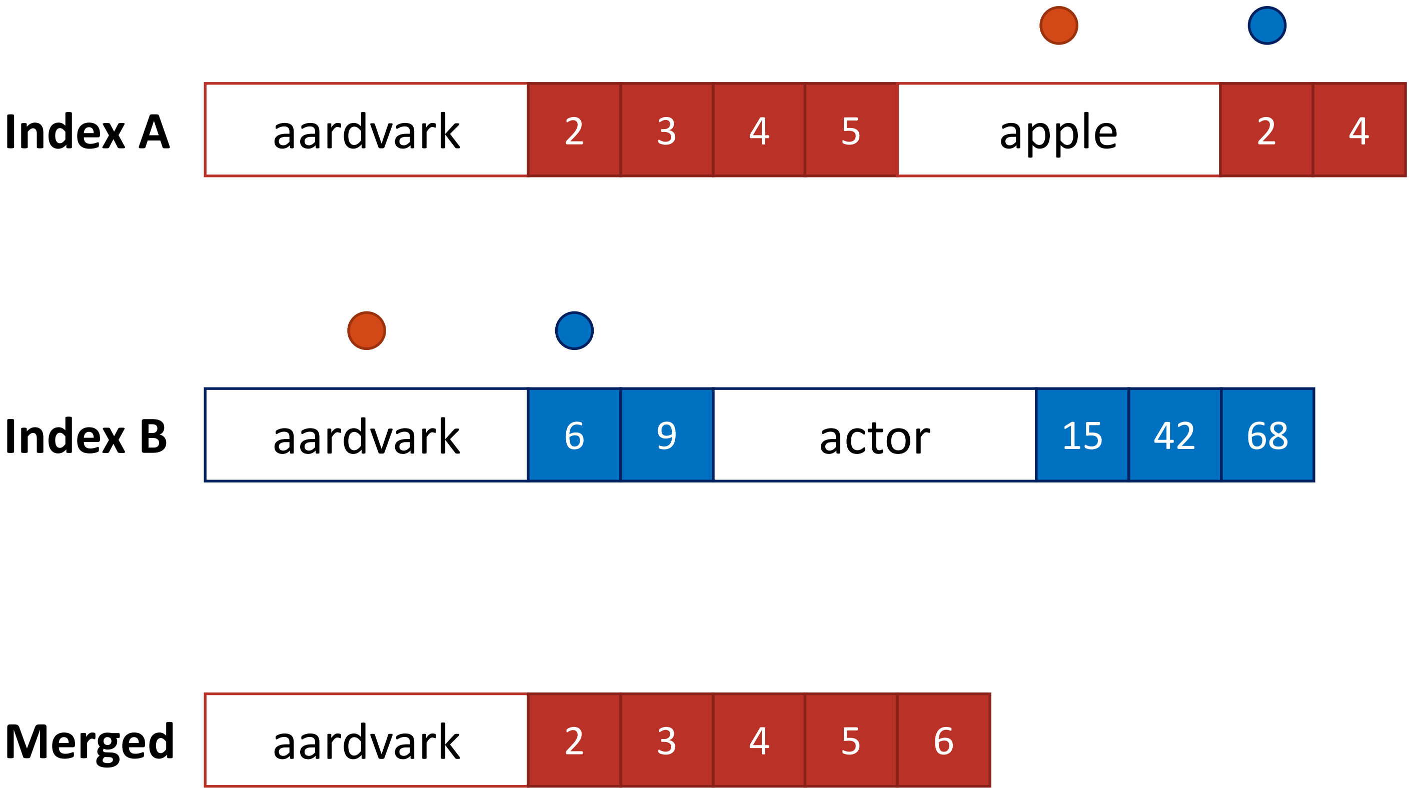
Internal merging



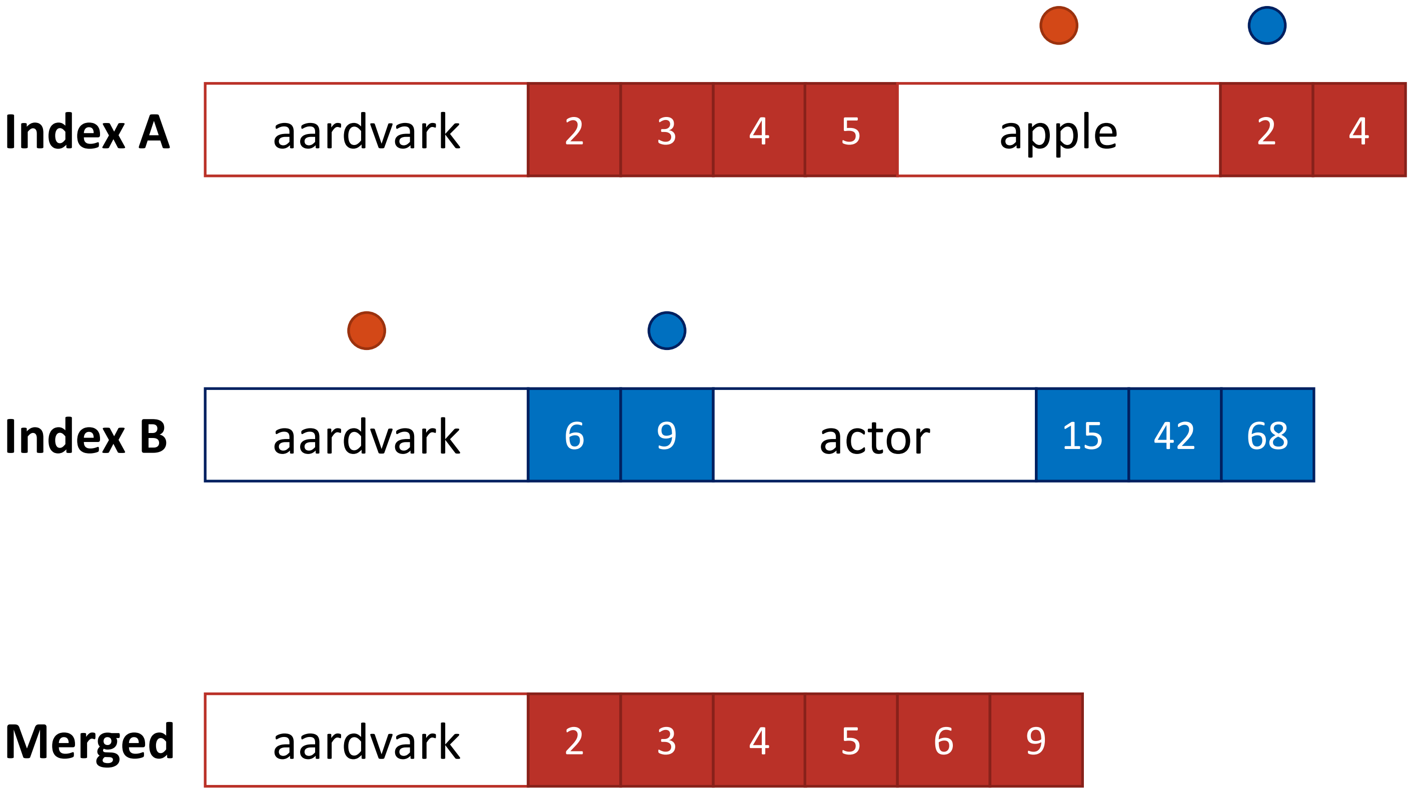
Internal merging



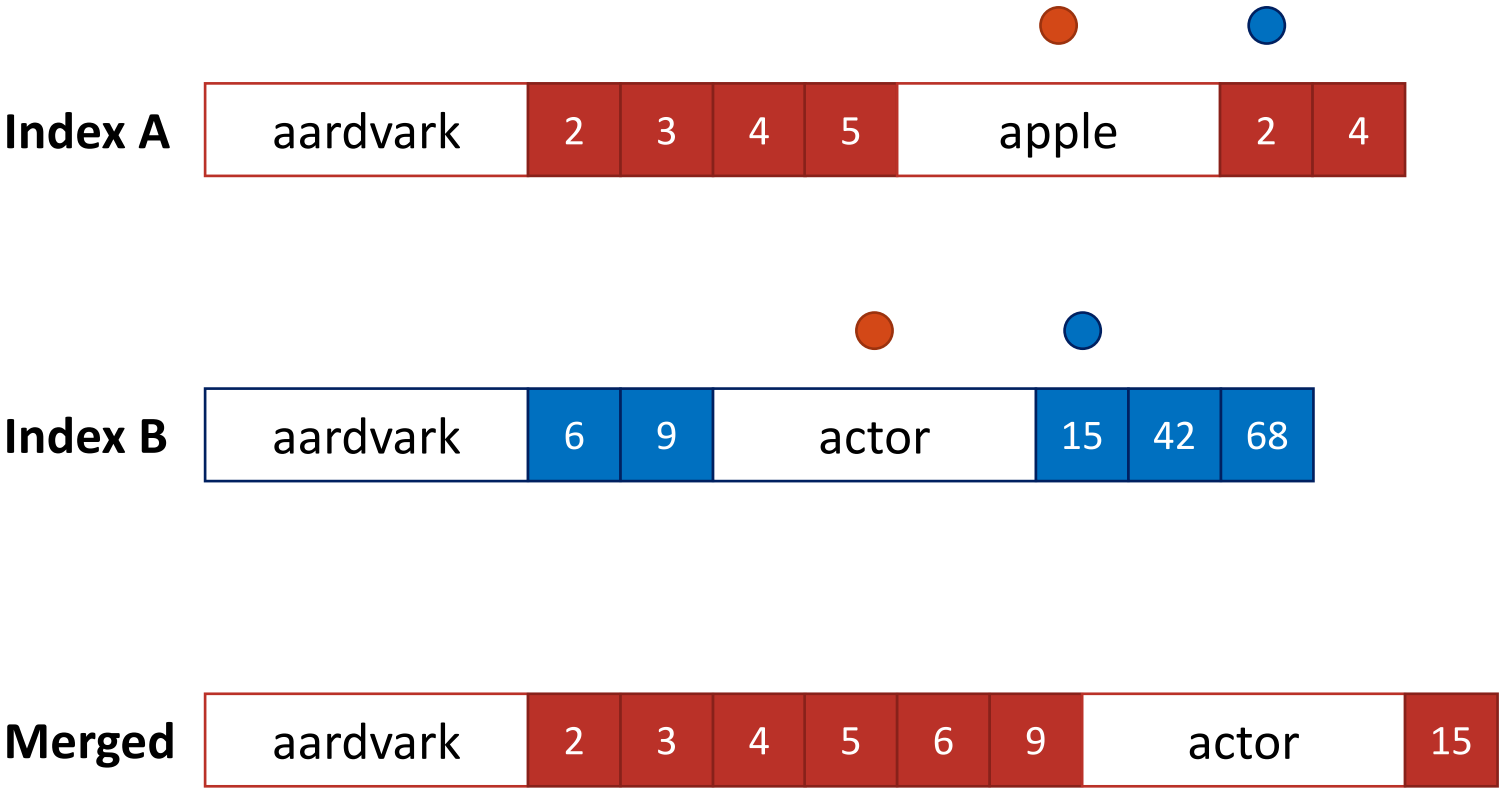
Internal merging



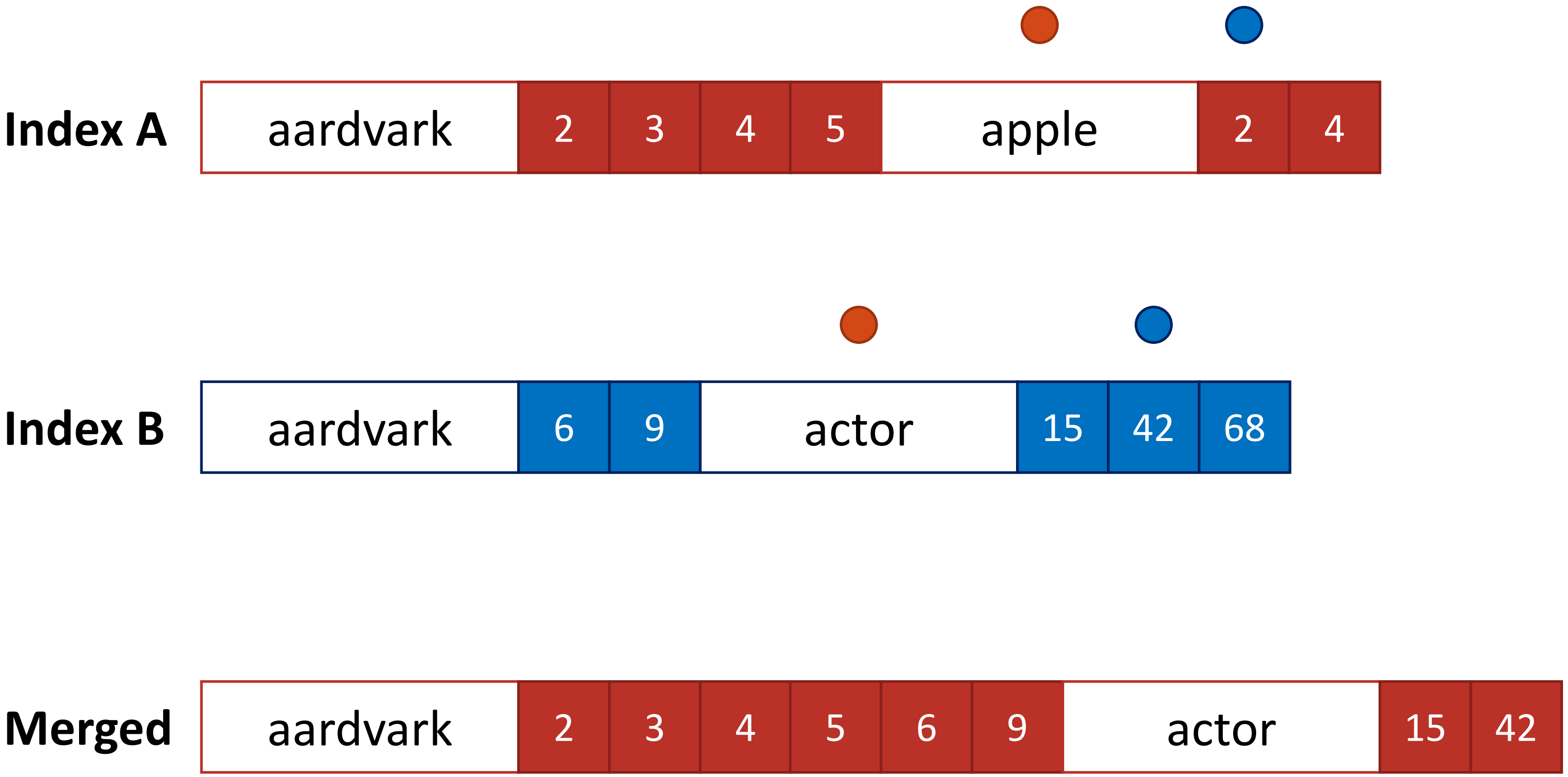
Internal merging



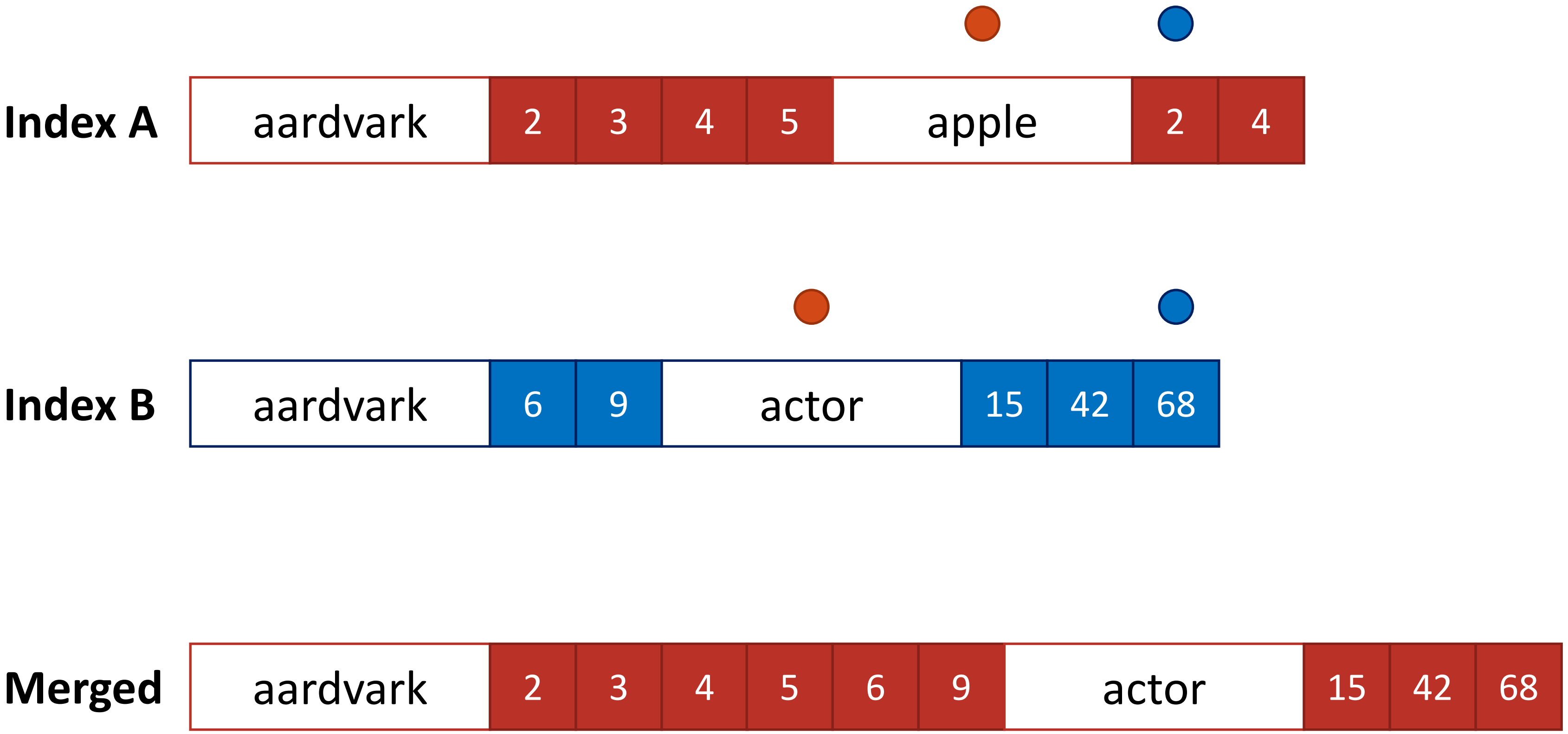
Internal merging



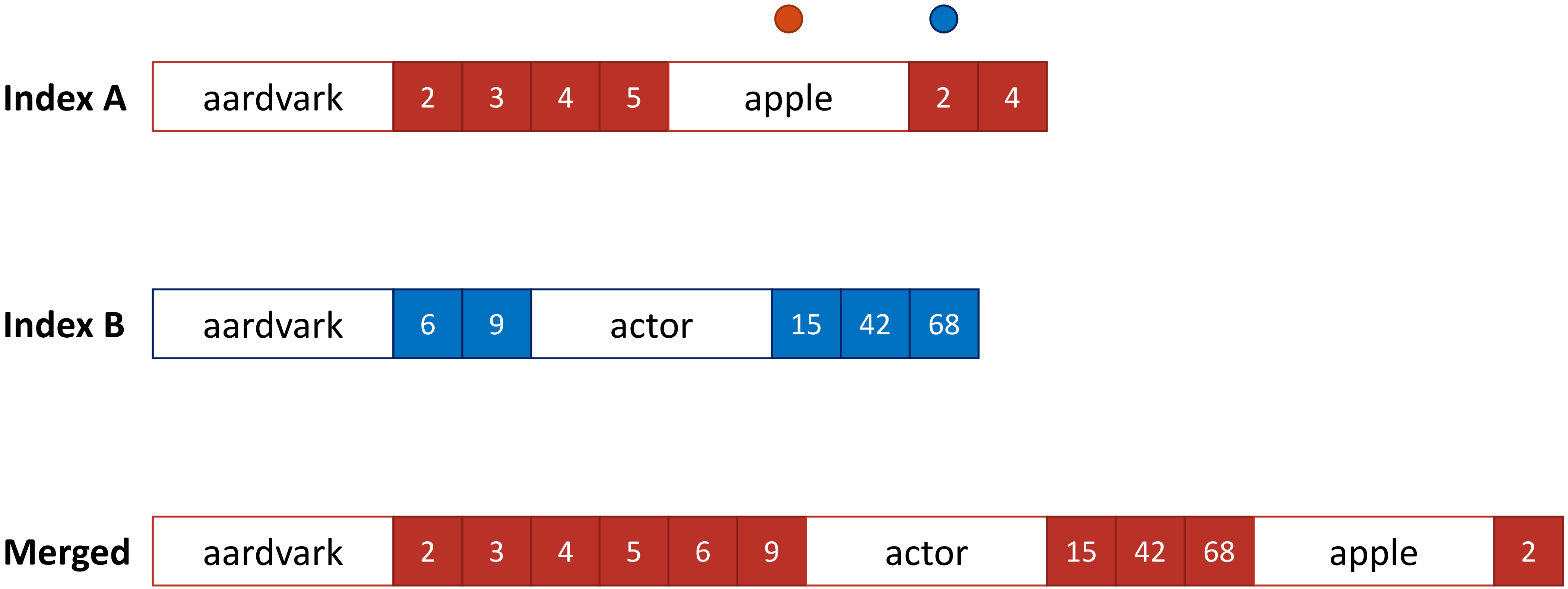
Internal merging



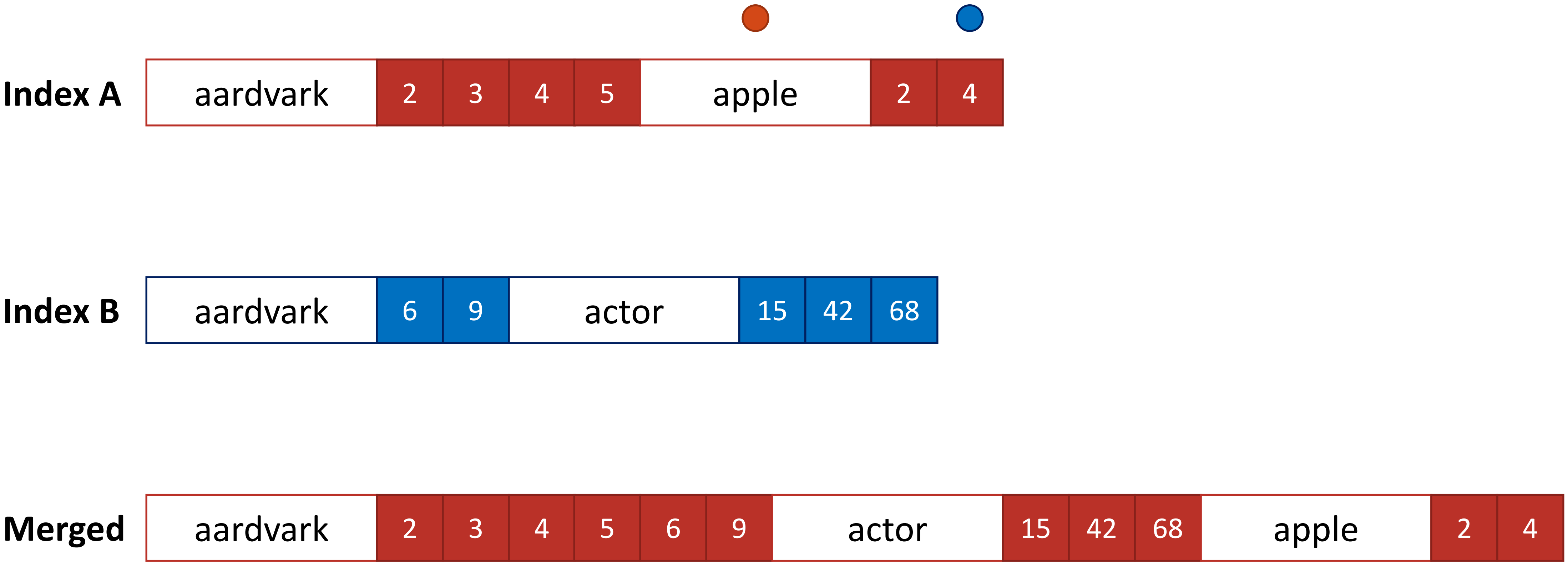
Internal merging



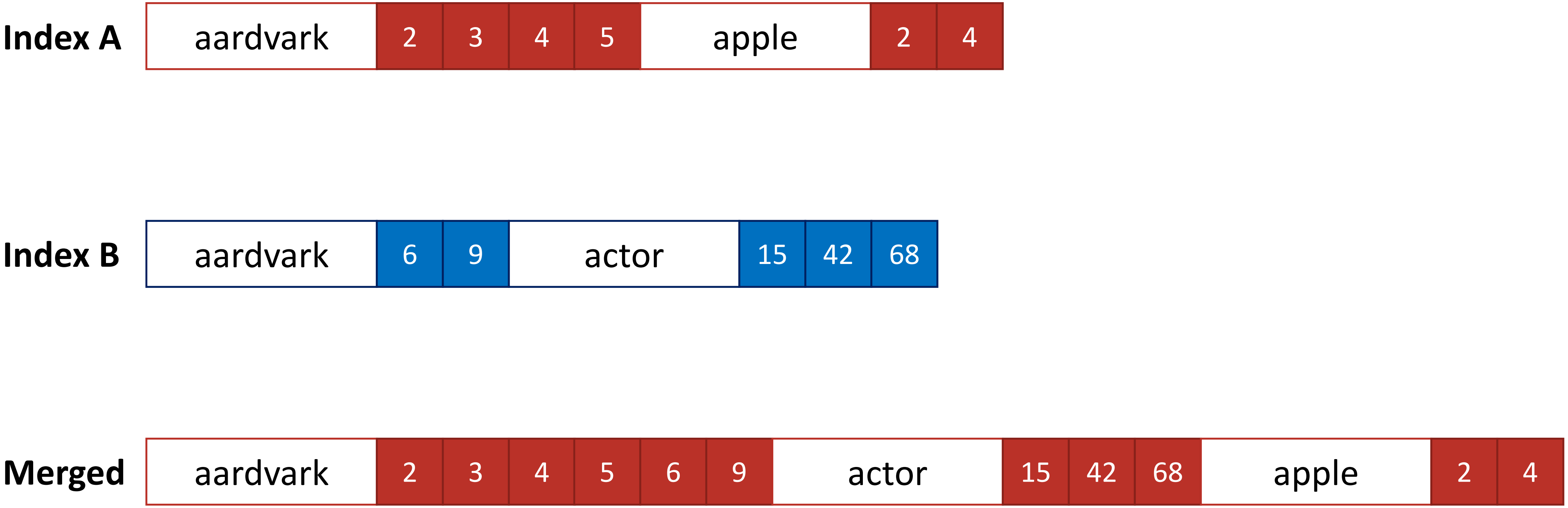
Internal merging



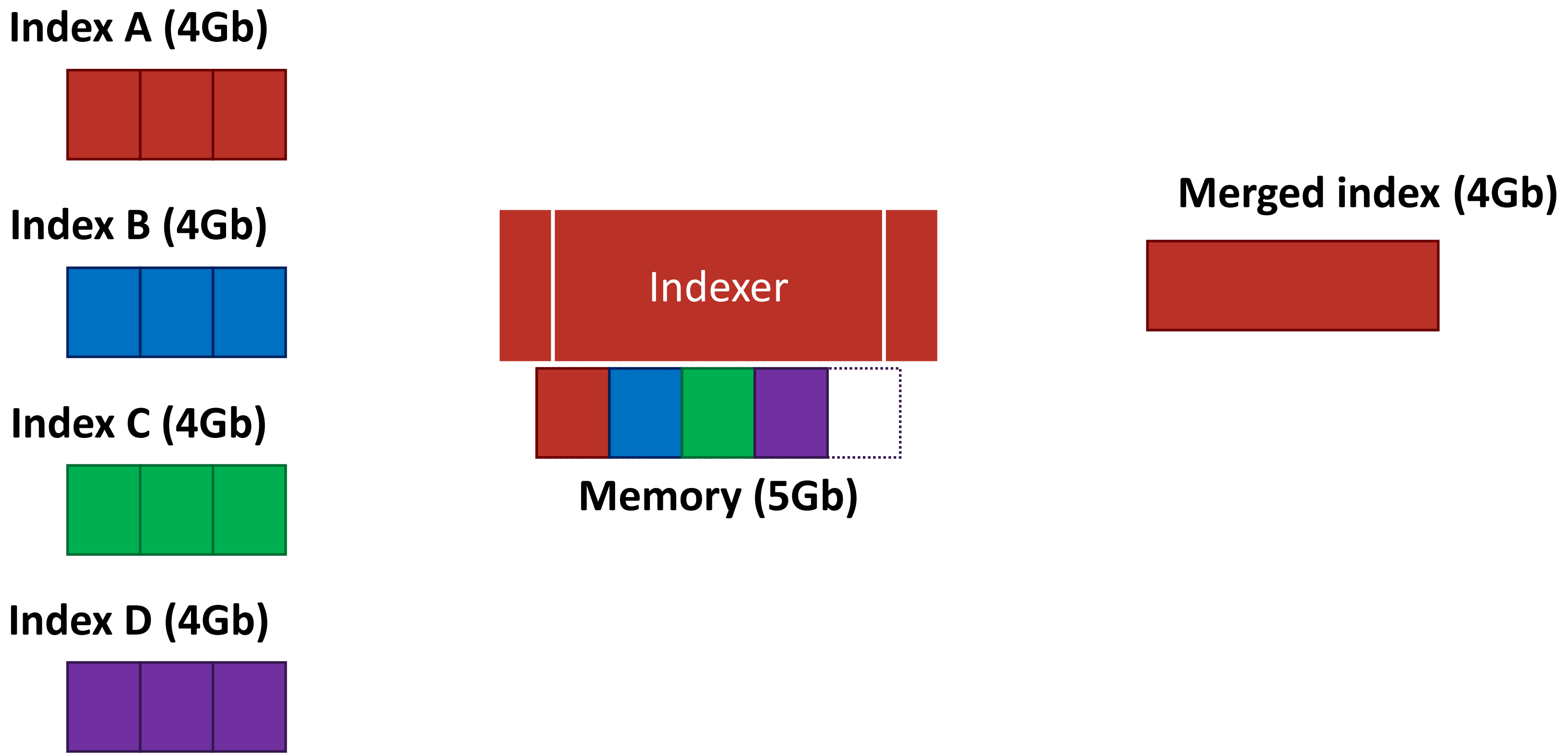
Internal merging



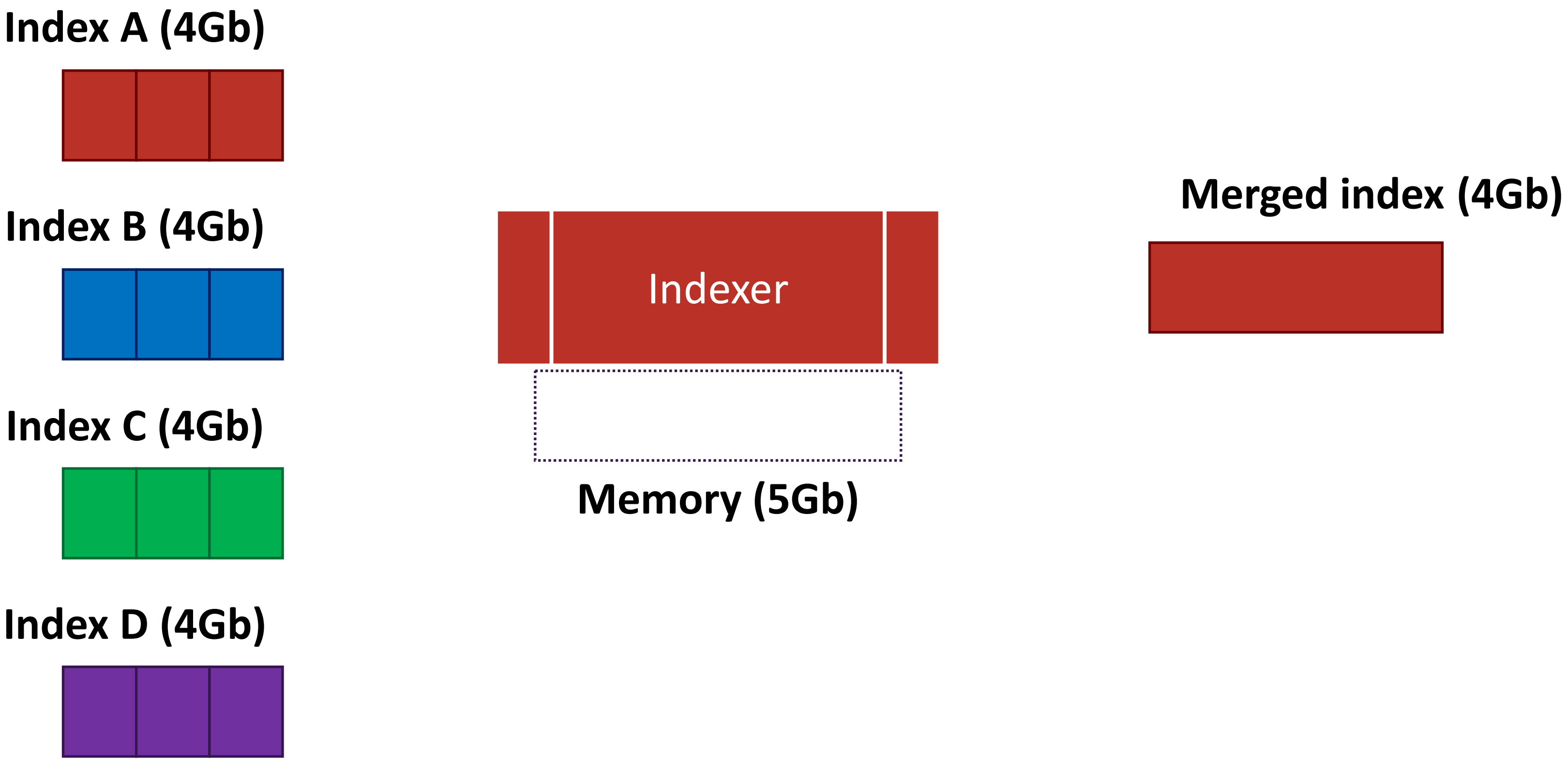
Internal merging



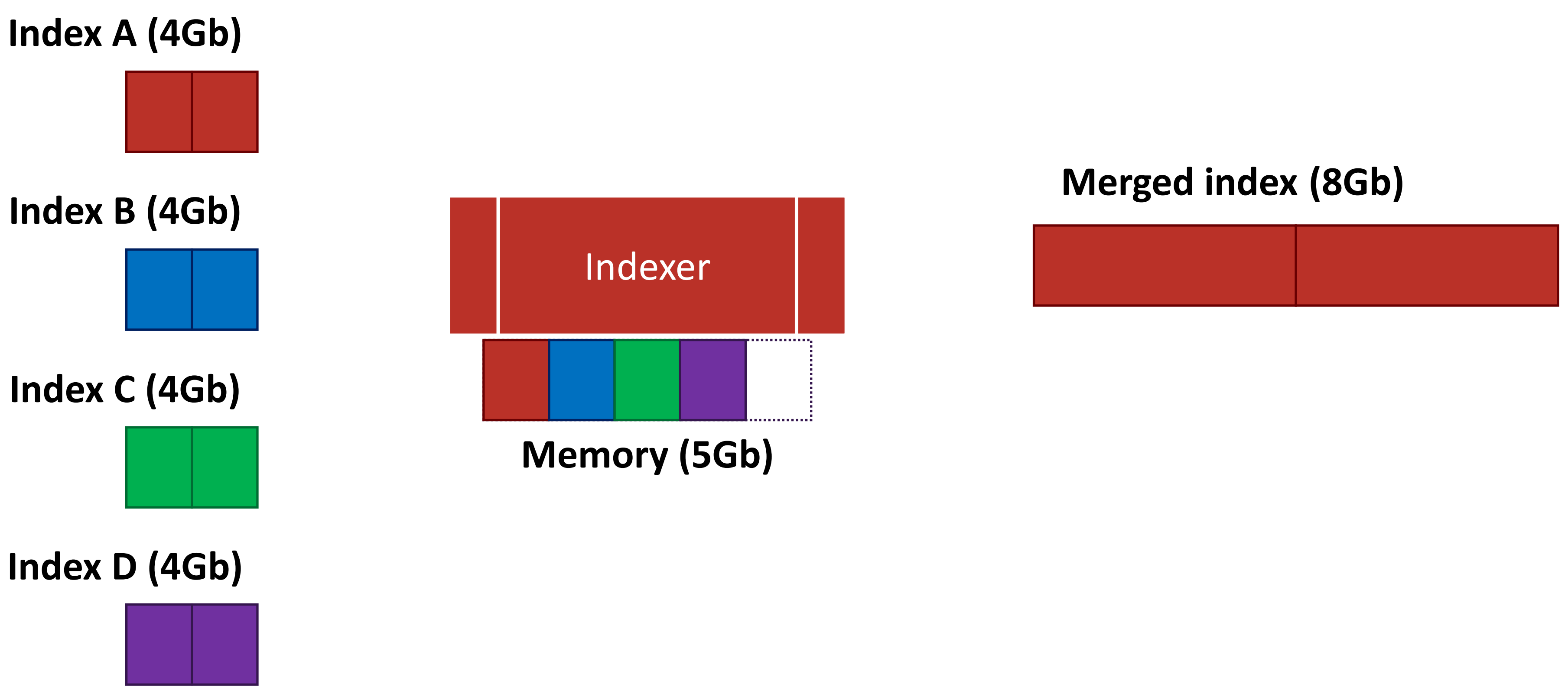
External merging



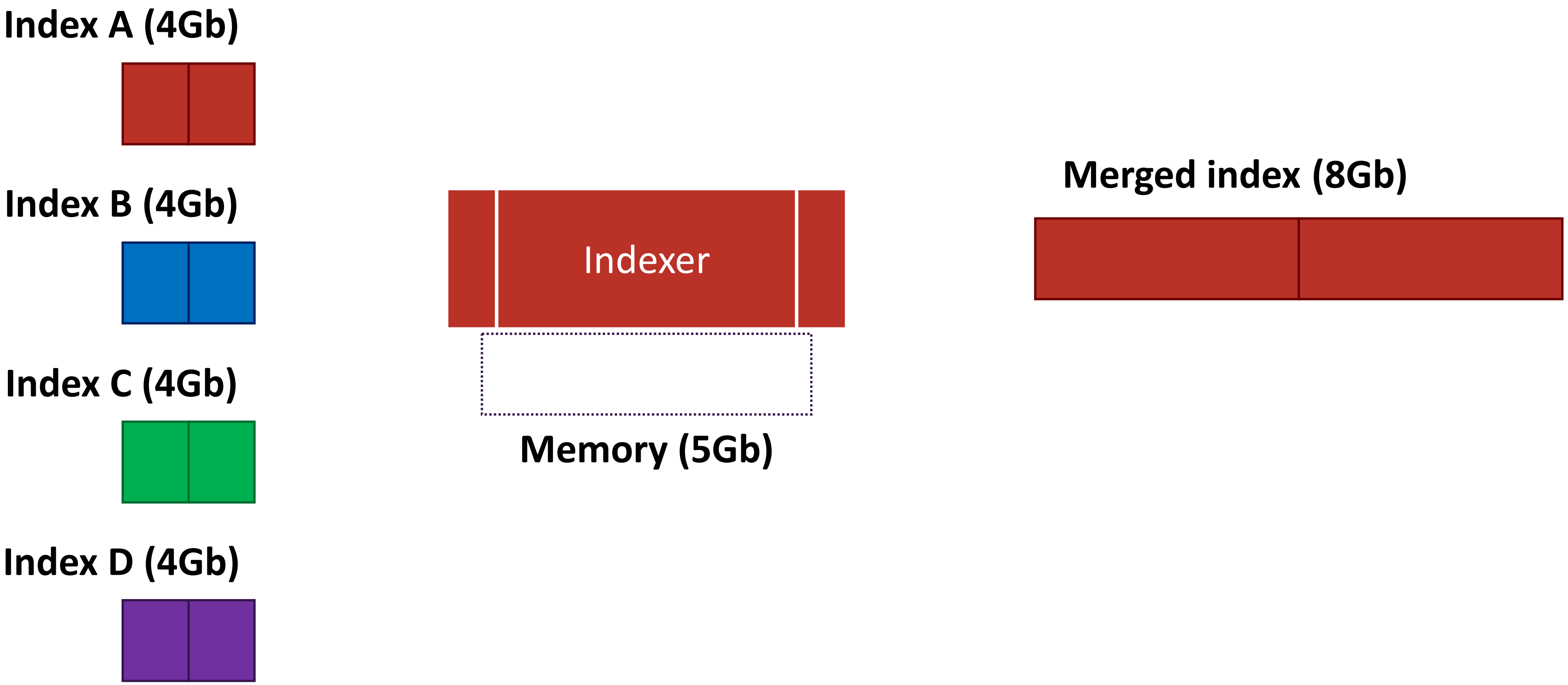
External merging



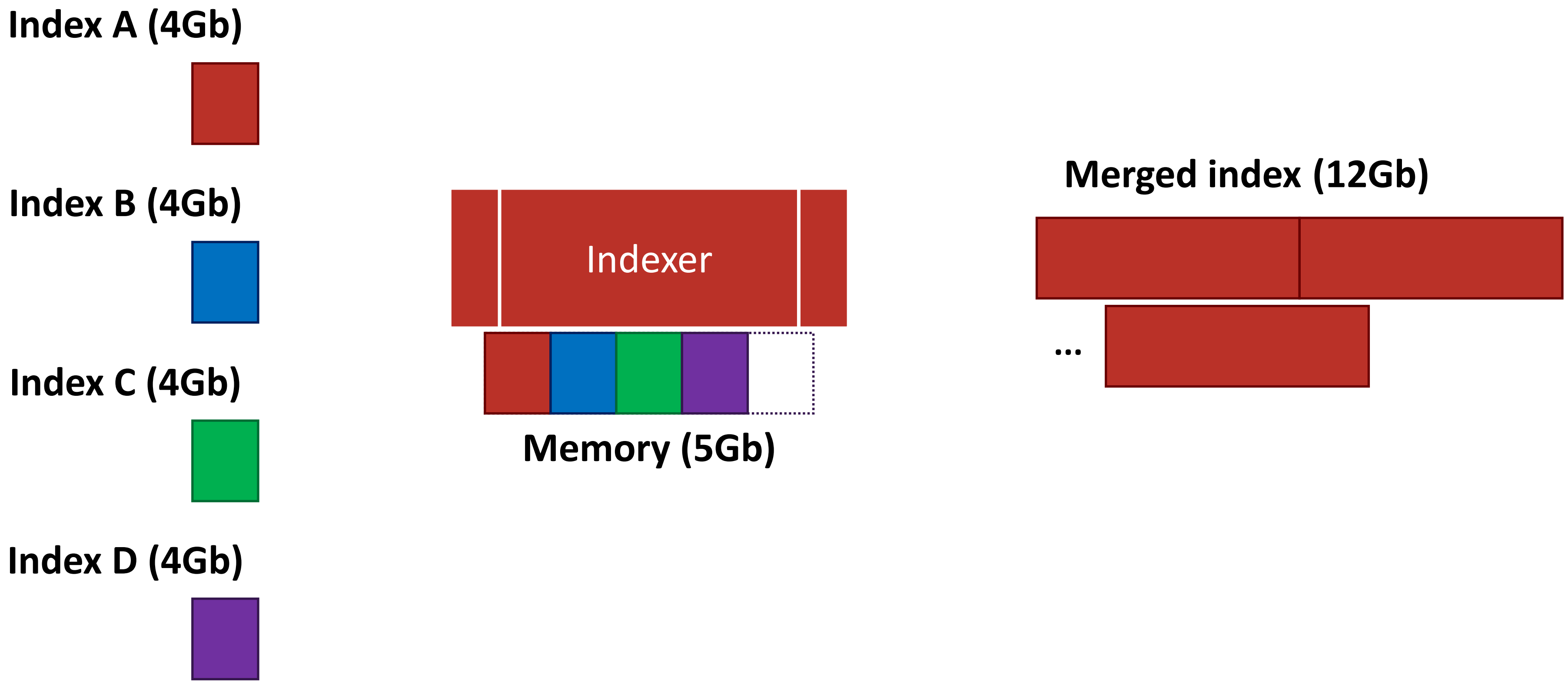
External merging



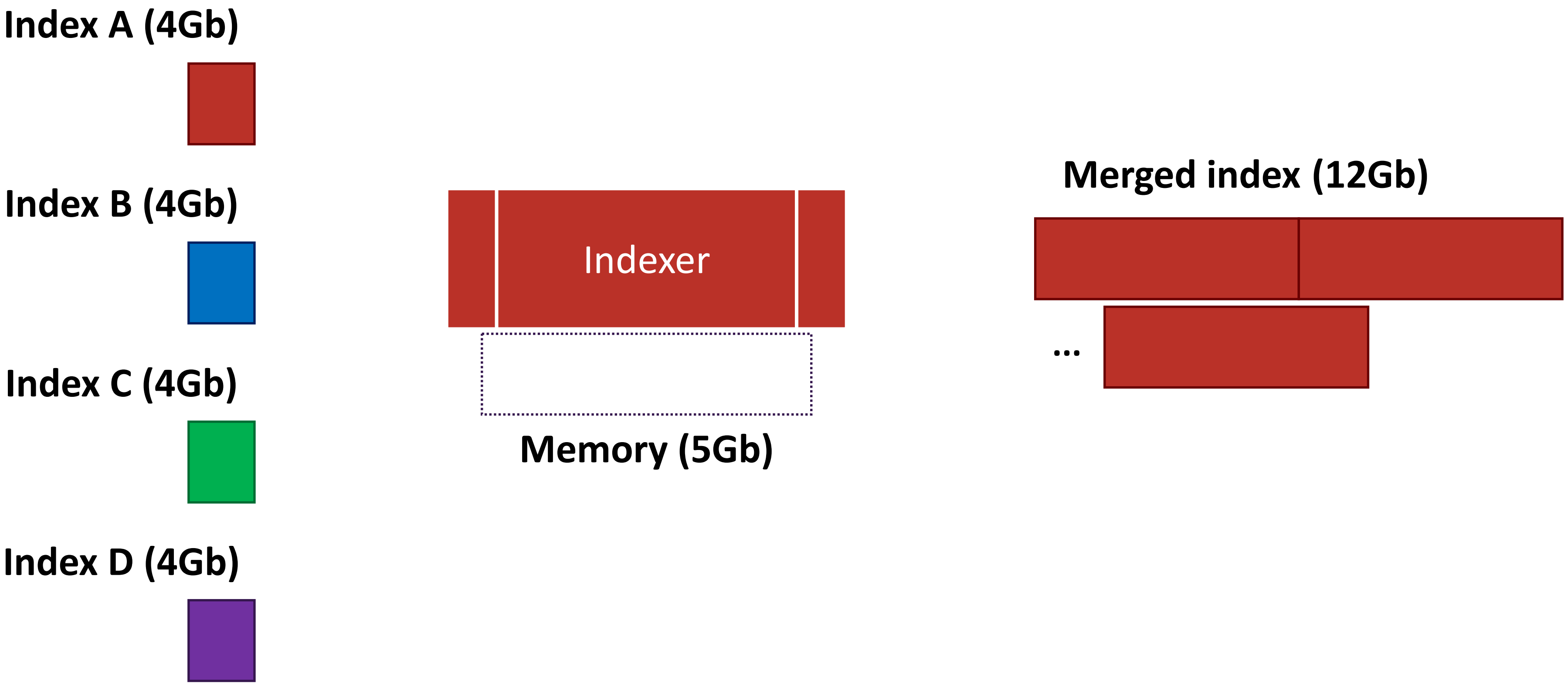
External merging



External merging



External merging



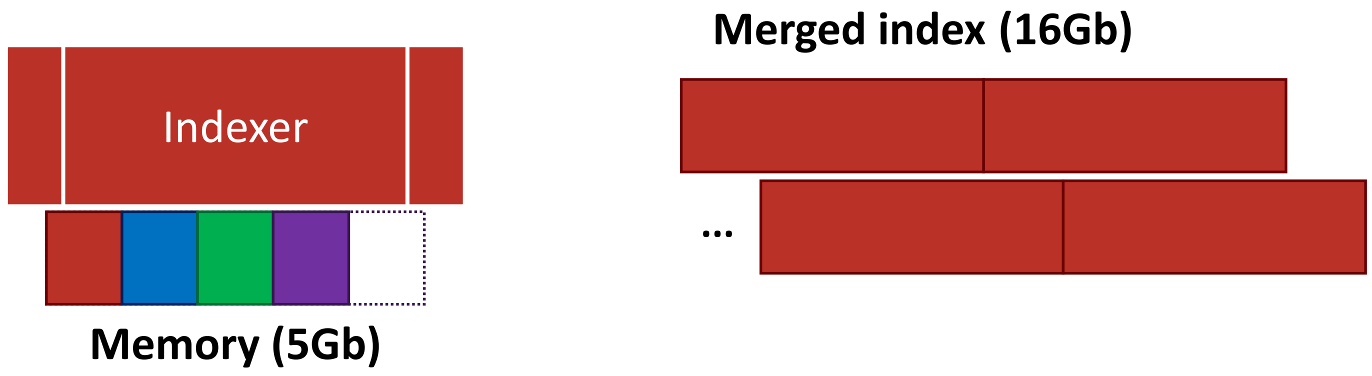
External merging

Index A (4Gb)

Index B (4Gb)

Index C (4Gb)

Index D (4Gb)



Index maintenance

Index merging efficient for large updates

- Overhead makes it inefficient for small updates

Instead, create separate index for new documents

- Merge *results* from both indexes

Could be in-memory, fast to update and search

- Also works for updates and deletions

Distributed indexing

Distributed processing driven by need to index and analyze huge amounts of data (i.e., the Web)

- Large numbers of inexpensive servers used rather than larger, more expensive machines

MapReduce is a distributed programming tool designed for indexing and analysis tasks

The MapReduce framework
[Dean and Ghemawat, USENIX 2004]

Key idea: keep data and processing together

- Bring code to data!

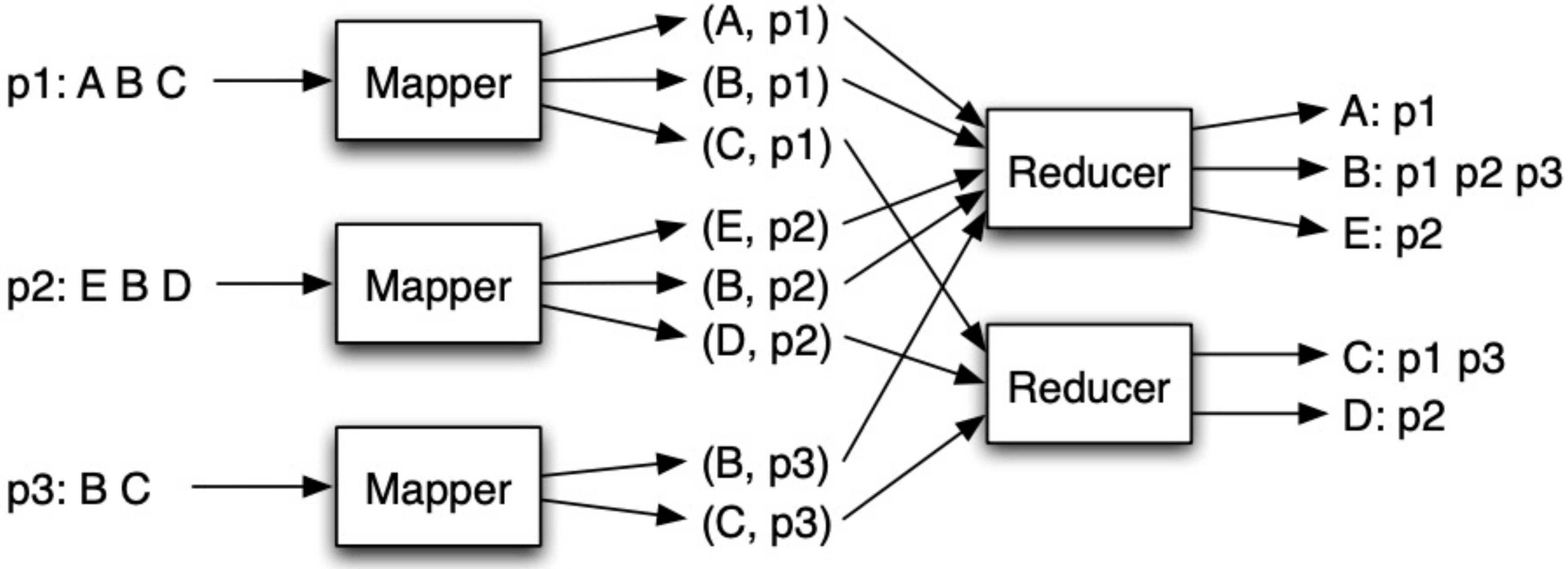
Map and reduce functions

- Inspired by functional programming

Constrained program structure

- But massive parallelization!

Distributed indexing with MapReduce
[McCreadie et al. IP&M 2012]



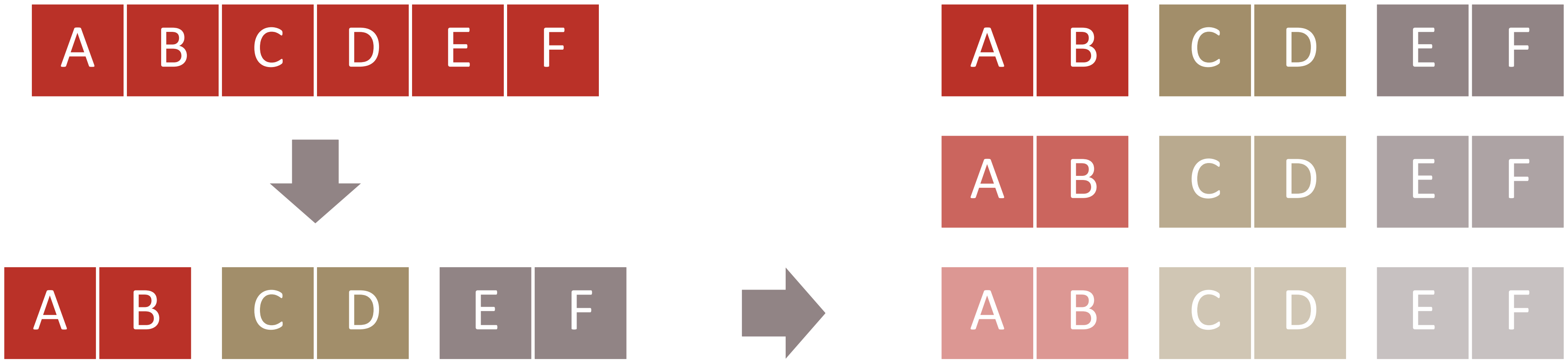
Index sharding and replication

Multiple index shards

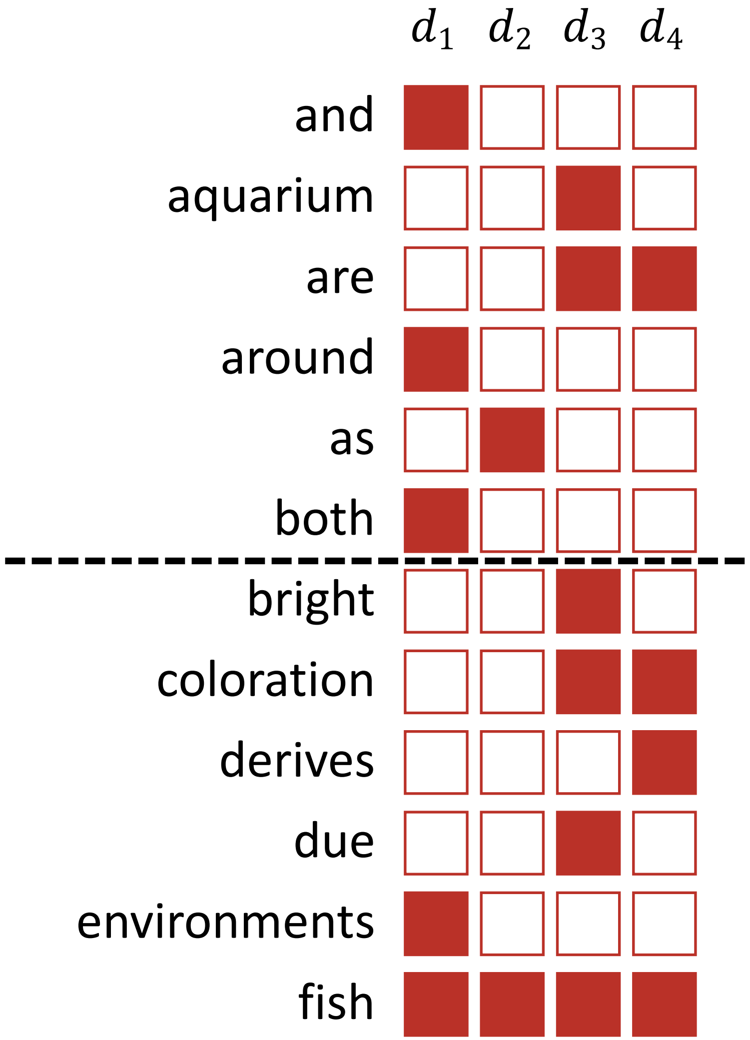
- Reduce query response times
- Scale with collection size

Multiple shard replicas

- Increase query throughput
- Scale with query volume
- Provide fault tolerance



Index sharding



Term-based sharding

- Disjoint terms in different nodes
- Single disk access per term

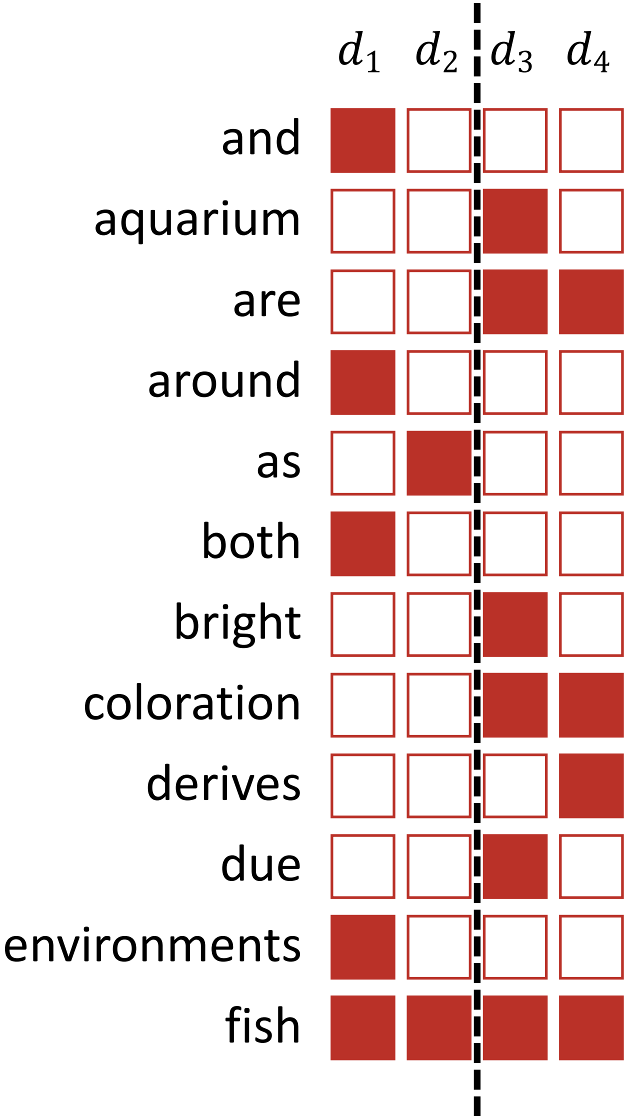
Good for inter-query parallelism

- Higher throughput (disjoint queries)

Bad for load balancing and resilience

- Nodes with popular terms overloaded
- Entire inverted lists missed on node failures

Index sharding



Document-based sharding

- Disjoint documents in different nodes
- Multiple (parallel) disk accesses per term

Good for intra-query parallelism

- Faster response (smaller indexes)
- Throughput increased via replication

Also good for load balancing and resilience

- Some documents missed on node failures

Summary

Indexing makes search feasible at a web-scale

- Effective search via carefully encoded postings
- Efficient search via carefully chosen structures

Several design choices

- Prefiltering, feature extraction, text understanding
- Posting design, ordering, compression, and more

Summary

Efficient management

- Memory-efficient, distributed construction
- Near-real time updates via clever merging
- Scalability via sharding and replication

References

Search Engines: Information Retrieval in Practice, Ch. 5
Croft et al., 2009

Scalability Challenges in Web Search Engines, Ch. 3
Cambazoglu and Baeza-Yates, 2015

UFMG UNIVERSIDADE FEDERAL DE MINAS GERAIS

Coming next...

Query Understanding

Rodrygo L. T. Santos
rodrygo@dcc.ufmg.br