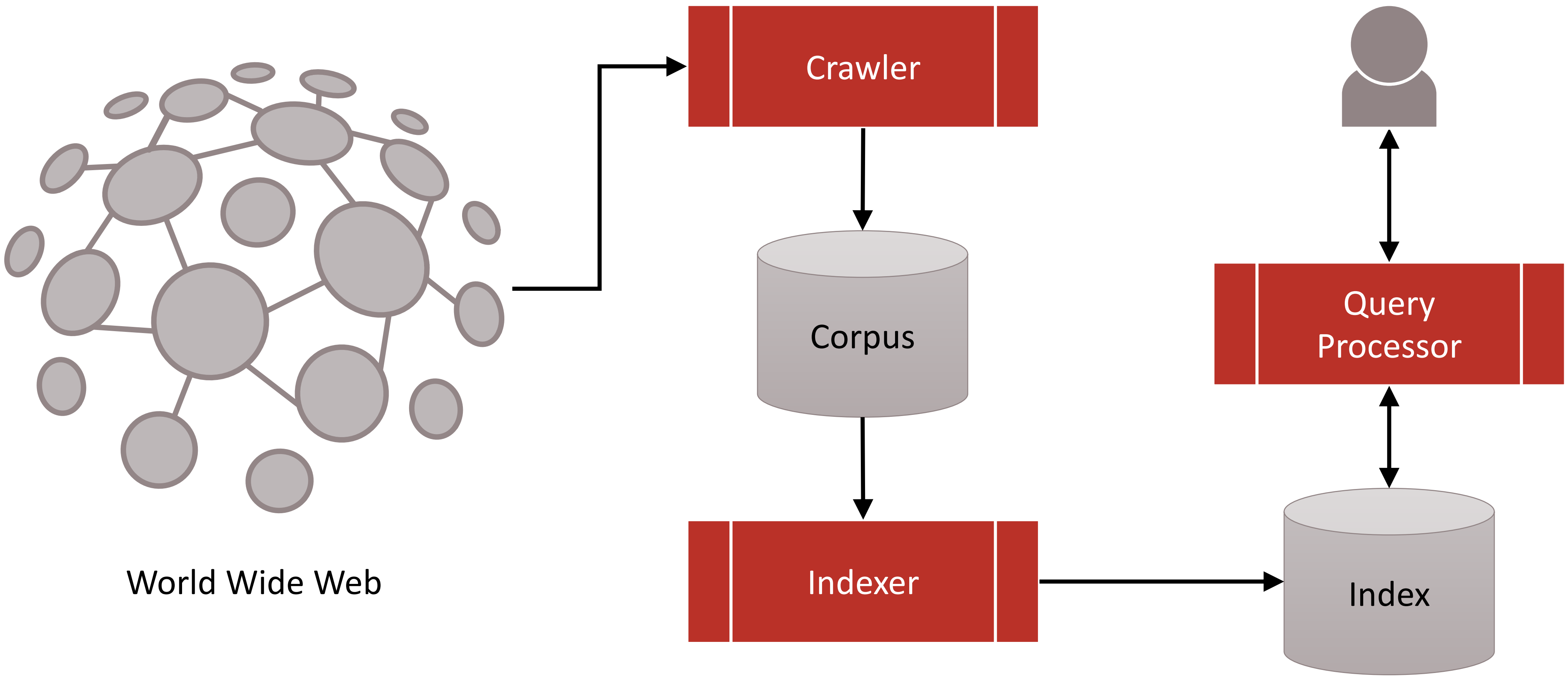
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Information Retrieval

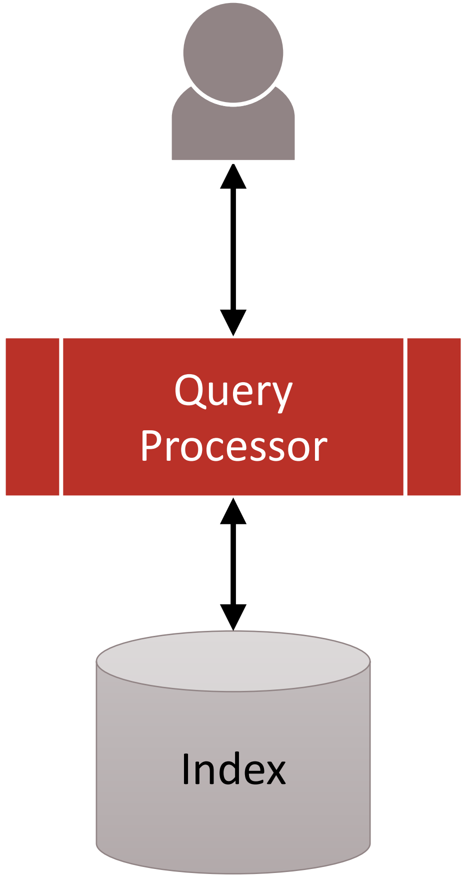
Document Matching

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

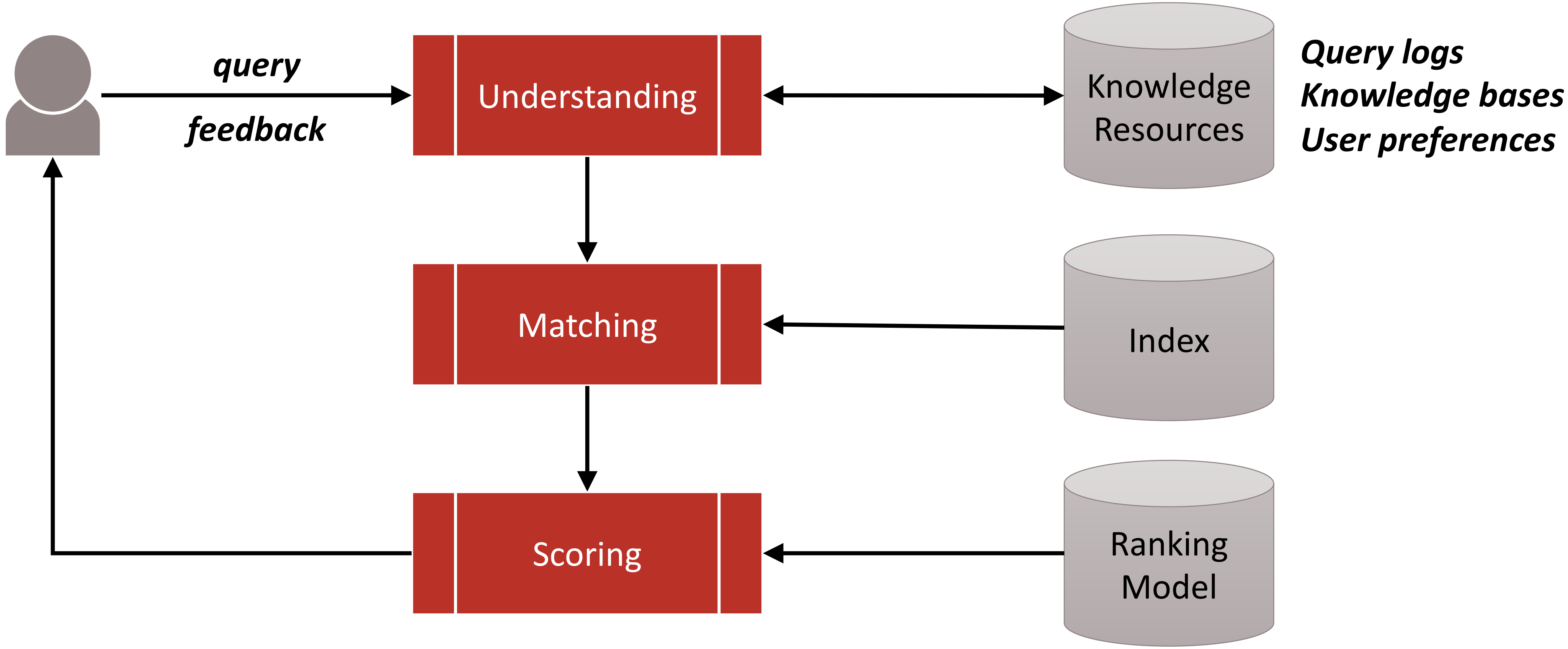
Search components



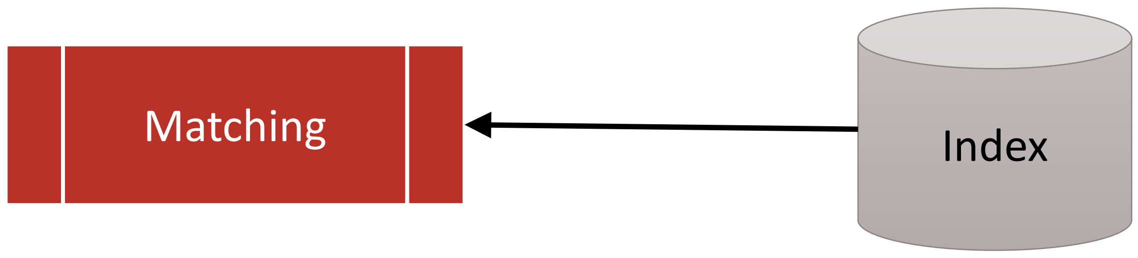
Search components



Query processing overview



Query processing overview



Document matching

Scan postings lists for all query terms

- [aquarium fish]

and	1:1			
aquarium	3:1			
are	3:1	4:1		
…				
environments	1:1			
fish	1:2	2:3	3:2	4:2

Document matching

Scan postings lists for all query terms

- [aquarium fish]

aquarium	3:1			
fish	1:2	2:3	3:2	4:2

Score matching documents

- $f(q,d) = \sum_{t \in q} f(t,d)$

Key challenge

Matching must operate under strict time constraints

- Even a slightly slower search (0.2s-0.4s) can lead to a dramatic drop in the perceived quality of the results

What makes it so costly?

- Must score billions (trillions?) of documents
- Must answer thousands of concurrent queries

Solution #1: bypass scoring

Query distributions similar to Zipf

- Popular queries account for majority of traffic

Caching can significantly improve efficiency

- Cache search results, or at least inverted lists

Problem: cache misses will happen eventually

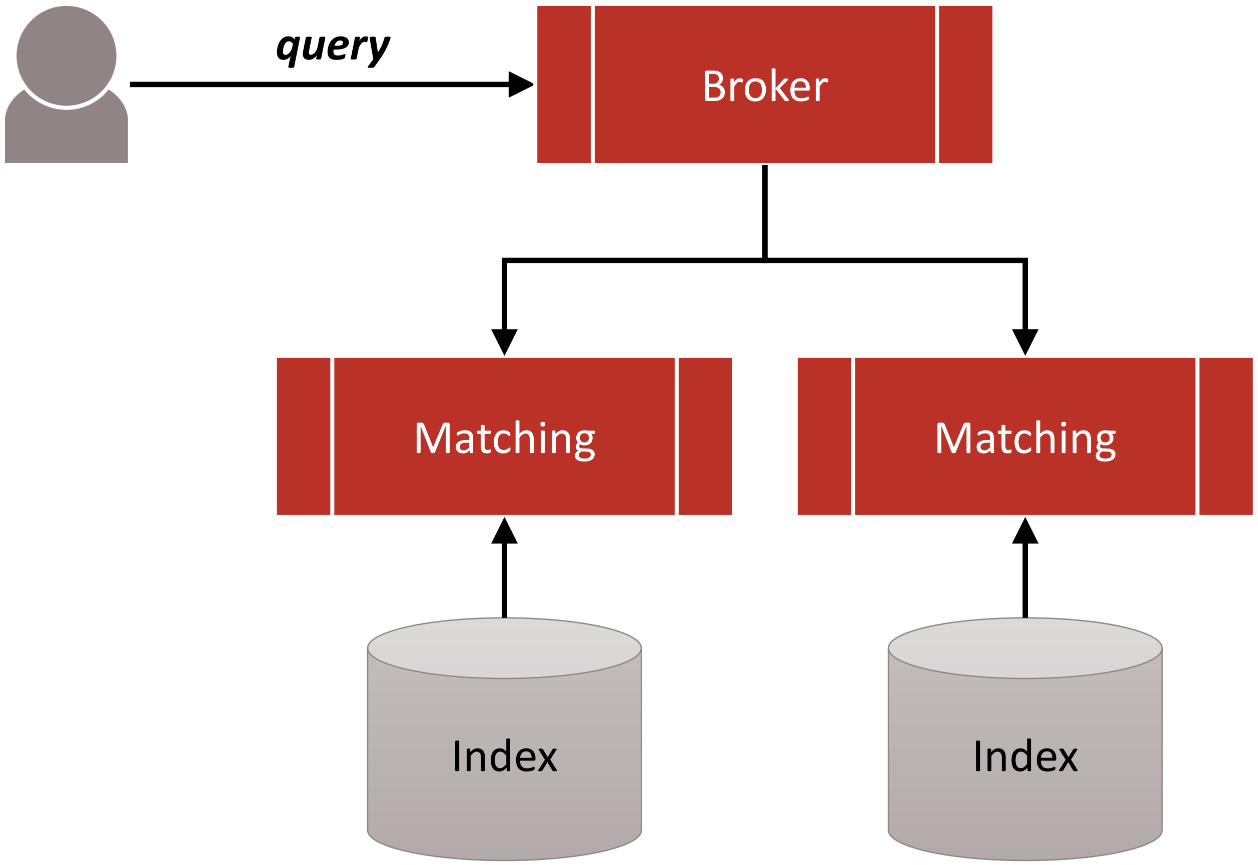
- New queries, index updates

Solution #2: distribute the burden

Indexes are often distributed in a cluster

- Too large to fit in a single machine
- Replication helps load balancing

Solution #2: distribute the burden



Solution #2: distribute the burden

Indexes are often distributed in a cluster

- Too large to fit in one machine
- Replication helps load balancing

Problem: cannot scale indefinitely

- Costly resources (hardware, energy)
- Intra-node efficiency still crucial

Solution #3: score parsimoniously

Some ranking models can be expensive

- Infeasible to score billions of documents

Ranking as a multi-stage cascade

- Stage #1: Boolean matching (billions)
- Stage #2: Unsupervised scoring (millions)
- Stage #3: Supervised scoring (thousands)

Why is it still so costly?

Inherent cost of matching documents to queries

- Query length (number of posting lists)
- Posting lists length (number of postings per list)

How to traverse postings?

Term-at-a-time (TAAT)

Inverted lists processed in sequence

- Partial document scores accumulated

salt	1:1	4:1	
water	1:1	2:1	4:1
tropical	1:2	2:2	3:1

Term-at-a-time (TAAT)

Inverted lists processed in sequence

- Partial document scores accumulated

salt	1:1	4:1	
water	1:1	2:1	4:1
tropical	1:2	2:2	3:1
score	1:1	4:1	

Term-at-a-time (TAAT)

Inverted lists processed in sequence

- Partial document scores accumulated

salt	1:1	4:1	
water	1:1	2:1	4:1
tropical	1:2	2:2	3:1
score	1:2	2:1	4:2

Term-at-a-time (TAAT)

Inverted lists processed in sequence

- Partial document scores accumulated

salt	1:1	4:1		
water	1:1	2:1	4:1	
tropical	1:2	2:2	3:1	
score	1:4	2:3	3:1	4:2

Term-at-a-time (TAAT)

```
function taat(query, index, k)
  scores = map()
  results = heap(k)
  for term in tokenize(query)
    postings = index[term]
    for (docid, weight) in postings
      if docid not in scores.keys()
        scores[docid] = 0
      scores[docid] += weight
  for docid in scores.keys()
    results.add(docid, scores[docid])
  return results
```

Document-at-a-time (DAAT)

Inverted lists processed in parallel

- One document scored at a time

salt	1:1	4:1	
water	1:1	2:1	4:1
tropical	1:2	2:2	3:1

Document-at-a-time (DAAT)

Inverted lists processed in parallel

- One document scored at a time

salt	1:1	4:1	
water	1:1	2:1	4:1
tropical	1:2	2:2	3:1
score	1:4		

Document-at-a-time (DAAT)

Inverted lists processed in parallel

- One document scored at a time

salt	1:1	4:1	
water	1:1	2:1	4:1
tropical	1:2	2:2	3:1
score	1:4	2:3	

Document-at-a-time (DAAT)

Inverted lists processed in parallel

- One document scored at a time

salt	1:1	4:1	
water	1:1	2:1	4:1
tropical	1:2	2:2	3:1
score	1:4	2:3	3:1

Document-at-a-time (DAAT)

Inverted lists processed in parallel

- One document scored at a time

salt	1:1	4:1		
water	1:1	2:1	4:1	
tropical	1:2	2:2	3:1	
score	1:4	2:3	3:1	4:2

Document-at-a-time (DAAT)

```
function daat(query, index, k)
  results = heap(k)
  targets = {docid for term in tokenize(query)
                   for docid in index[term]}
  lists = [index[term] for term in tokenize(query)]
  for target in targets
    score = 0
    for postings in lists
      for (docid, weight) in postings
        if docid == target
          score += weight
    results.add(target, score)
  return results
```

Optimization techniques

No clear winner

- TAAT is more memory efficient (sequential access)
- DAAT uses less memory (no accumulators)

Naïve versions can be improved

- Calculate scores for fewer documents (conjunctions)
- Read less data from inverted lists (skipping)

Matching semantics

Disjunctive matching

- Documents must contain at least one query term
- More matches, lower precision

Problem: all lists must be fully traversed

- There may be a relevant document in the last position of a (very long) posting list for a common term

Matching semantics

Conjunctive matching

- Documents must contain all query terms
- Fewer matches, higher precision

Conjunctive matching preferred in practice

- Effective and efficient for short queries
- Combined with relaxation for long queries

Skipping

Search involves comparing lists of different lengths

- Linear scanning can be very inefficient

Skip ahead to document under comparison

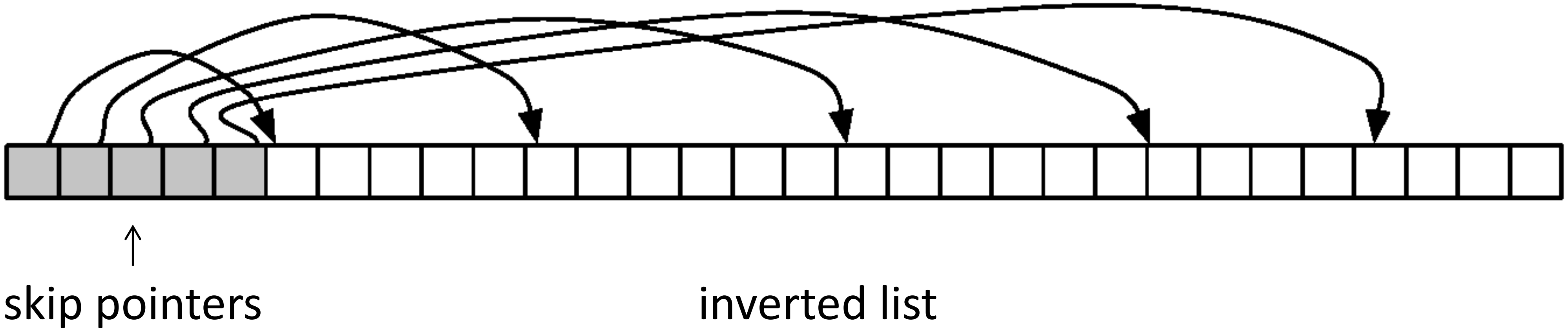
- Compression makes jumping difficult
  (variable size encodings, only d-gaps stored)

Solution: store skip pointers in the index

Skip pointers

A skip pointer (d, p) for document d and position p

◦ Docid d precedes posting at (zero-based) position p



Skipping example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
inverted list	5	11	17	21	26	34	36	37	45	48	51	52	57	80	89
delta gaps	5	6	6	4	5	9	2	1	8	3	3	1	5	23	9
skip pointers	(17, 3)		(34, 6)		(45, 9)		(52, 12)								

How to skip to docid 37?

Skipping example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
inverted list	5	11	17	21	26	34	36	37	45	48	51	52	57	80	89
delta gaps	5	6	6	4	5	9	2	1	8	3	3	1	5	23	9

skip pointers	(17, 3)	(34, 6)	(45, 9)	(52, 12)

How to skip to docid 54?

Other approaches

Unsafe early termination

- Ignore high-frequency word lists in TAAT
- Ignore documents at end of lists in DAAT

Can be improved with index tiering

- Postings ordered by quality (e.g., PageRank)
- Postings ordered by score (e.g., BM25)

Summary

Document matching can be challenging

- Traversing multiple, long inverted lists

Several complementary approaches

- Caching, distribution, cascading

Efficient index traversal still crucial

- Classic techniques, several optimizations

References

Search Engines: Information Retrieval in Practice, Ch. 5

Croft et al., 2009

Scalability Challenges in Web Search Engines, Ch. 4

Cambazoglu and Baeza-Yates, 2015

Efficient Query Processing Infrastructures

Tonellotto and Macdonald, SIGIR 2018

UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Coming next...

Efficient Matching

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