

Efficient In-Memory Inverted Indexes

Theory and Practice



Joel Mackenzie
The University of
Queensland



Sean MacAvaney
University of
Glasgow



Antonio Mallia
Pinecone



Michał Siedlaczek
MongoDB, Inc.



SIGIR 2025 – Padua, Italy – July 2025



Who are we?



Joel Mackenzie
The University of
Queensland



Sean MacAvaney
University of
Glasgow



Antonio Mallia
Pinecone



Michał Siedlaczek
MongoDB, Inc.

**Early Career Academics/Practitioners
interested in efficient and effective IR
systems. All very keen on empirical,
“hands-on” research and development.**

<https://jmmackenzie.io/>
<https://macavaney.us/>
<https://www.antoniomallia.it/>
<https://siedlaczek.me/>

Who are you?

- **Newcomers** with no assumed knowledge of efficient inverted index-based architectures and query processing.
- **Experienced Researchers** who are keen to sharpen their skills and improve their understanding of efficient indexing and retrieval.
- **Everyone in between!**

Why should you care?

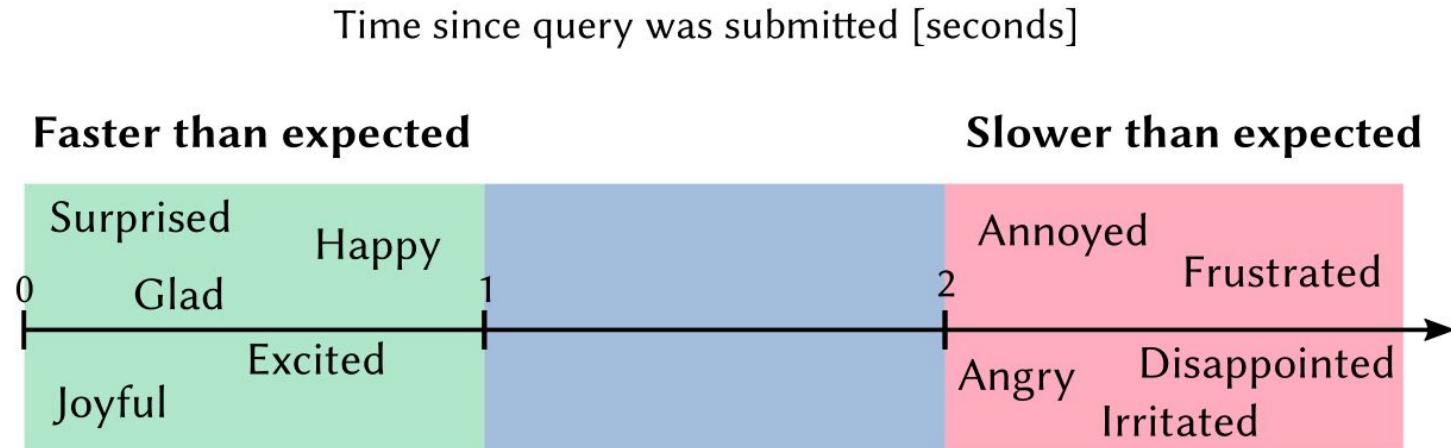
- **Despite the best efforts** of the IR community, inverted indexes just will not die!
 - Sometimes you really do need to find documents that contain a specific set of terms;
 - Traditional ranking models like BM25 continue to be a strong baseline on unseen data;
 - Inverted indexes tend to scale extremely well as collections grow large.

Why should you care?

- **Next generation** ranking methods such as *learned sparse retrieval* are still making use of inverted indexes.
- **Possessing a strong theoretical understanding** of inverted indexes, including how they can be engineered to be efficient in practice, is still very relevant in 2025!
- This tutorial will cover both the theory and practice necessary to understand, experiment with, and contribute to the future of inverted indexes.

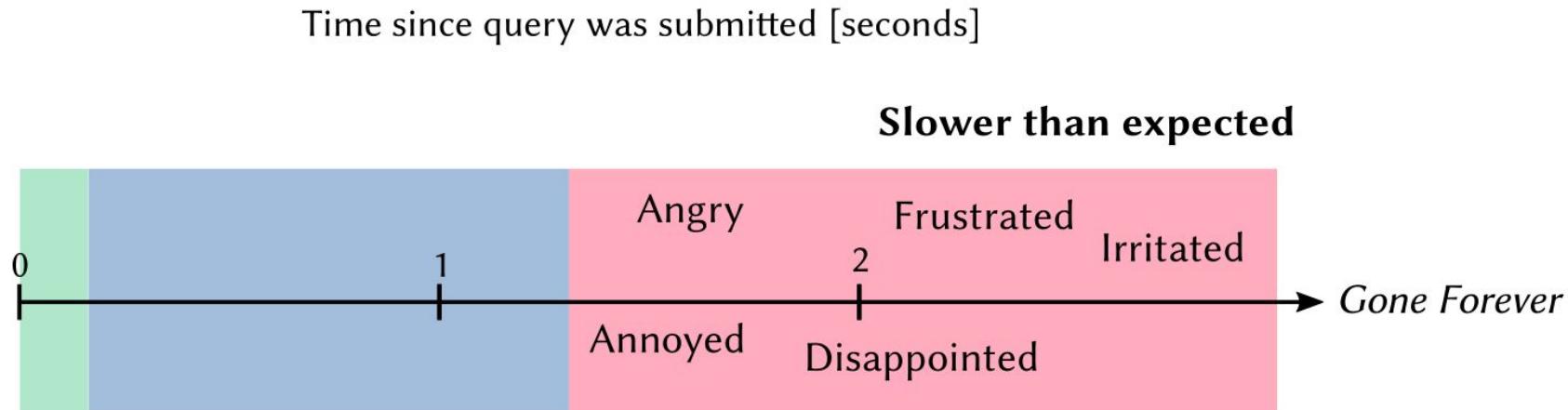
Large-Scale IR

“returning good results quickly is better than returning the best results slowly”
– Dean and Barosso, CACM, 2013.



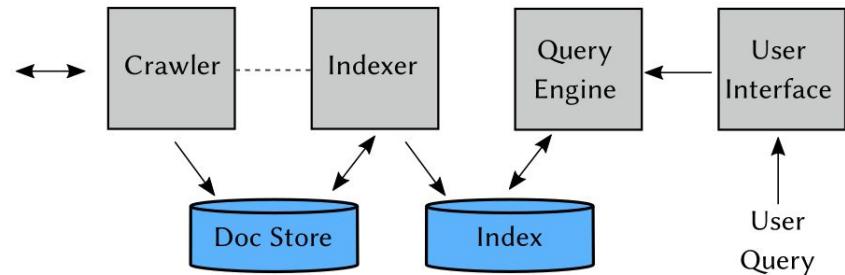
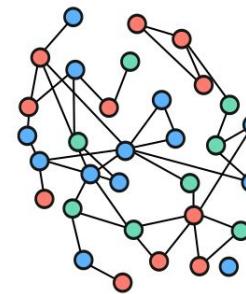
Large-Scale IR

“returning good results quickly is better than returning the best results slowly”
– Dean and Barosso, CACM, 2013.



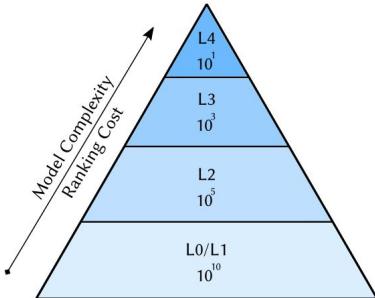
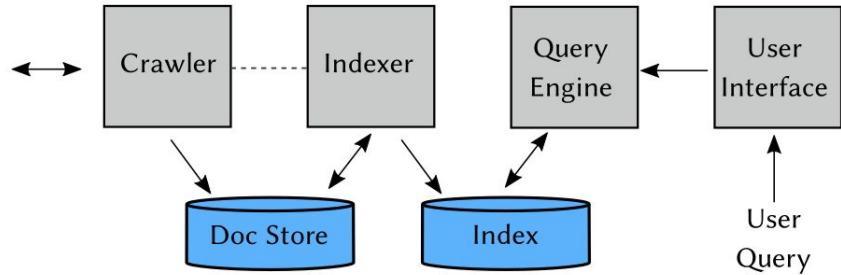
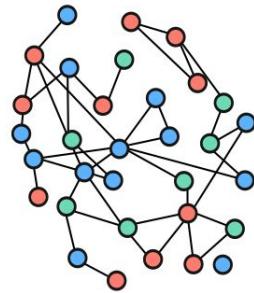
Large-Scale IR

World Wide Web



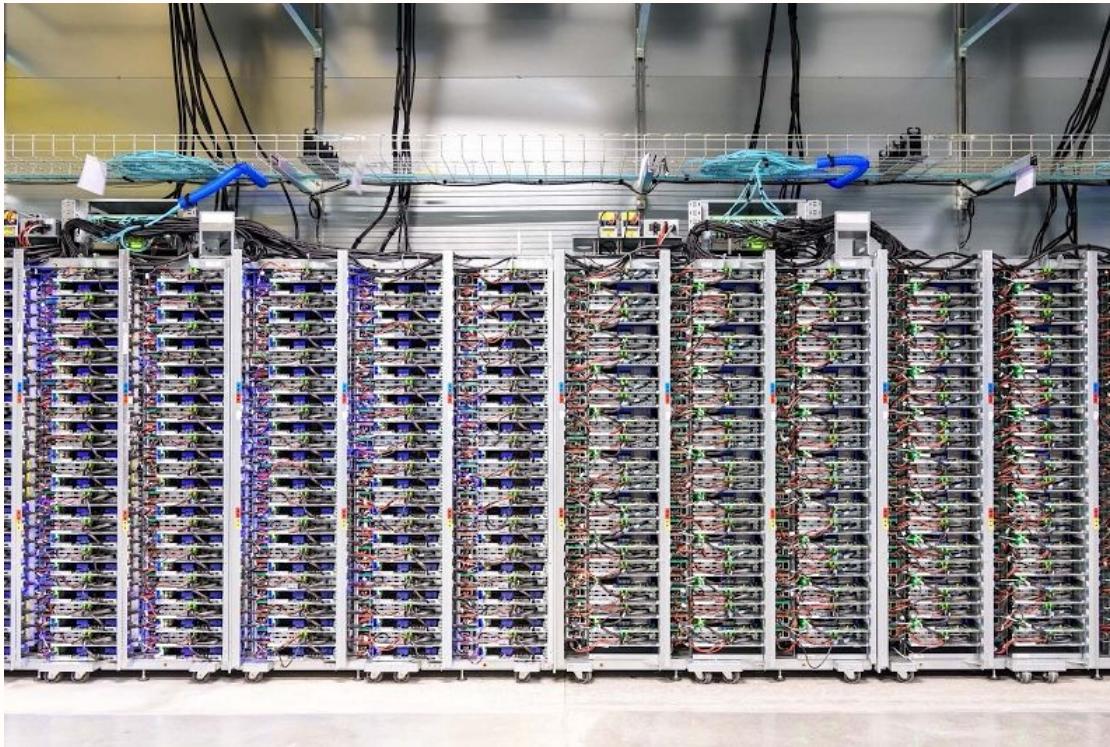
Large-Scale IR

World Wide Web



Candidate Generation

Large-Scale IR



<https://www.google.com/about/datacenters/gallery>

Today's Plan

Session 1: (Basic) Indexing and Retrieval 09.00 – 10.30

Morning Tea 10.30 – 11.00

Session 2: Learned Sparse Retrieval 11.00 – 12.00

Session 3: New Directions 12.00 – 12.30

Intended Learning Outcomes

ILO 1: Theoretical Understanding of Inverted Indexes

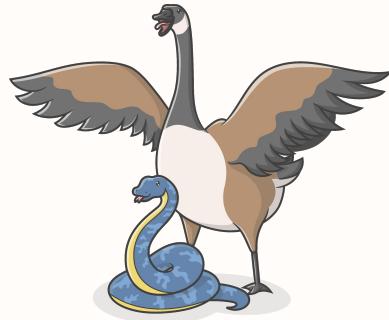
ILO2: Fast Top- k Retrieval with Dynamic Pruning

ILO3: Current Trends and Research Directions

ILO4: Experimenting with PISA

ILO5: Integrating PISA into Modern Applications

IR Experimentation



History and Origins of PISA

The **PISA engine** started off as the *Data Structures for Inverted Indexes (ds2i)* project in 2014.

It formed the basis of “*Partitioned Elias Fano Indexes*” by Ottaviano and Venturini, which won the **SIGIR 2014** best paper award, as well as “*Optimal Space-Time Tradeoffs for Inverted Indexes*” by Ottaviano, Tonelotto, and Venturini, **WSDM 2015**.

In 2017, “*Faster BlockMax WAND with variable-sized blocks*” was published at **SIGIR** by Mallia, Ottaviano, Porciani, Tonelotto, and Venturini. At this point, **PISA was forked from ds2i**.

The PISA Engine

An efficient, extensible, modern search engine.



- Written in C++17
- In-memory retrieval
- Low-level optimization out-of-the-box: CPU intrinsics, branch prediction hinting, ...
- Extensible: Plug and play parsing, stemming, compression, query processing
- Indexing, parsing, sharding capabilities
- Free, open-source permissive license

Where PISA Shines

Ridiculously fast top- k query processing



Extensible experimentation with easy access to state-of-art methods

Small but active group of maintainers

Interfaces well with other experimental IR systems

Where PISA Pales

Primary focus is on bag-of-words, top- k retrieval



No support for positional indexing, fields, ...

Not so user friendly due to high complexity of the codebase

> See: https://github.com/terrierteam/pyterrier_pisa for a higher-level Python API



PISA's role in the IR Ecosystem



TheRealMasonMac • 4y ago

In the benchmarks, how is Pisa so fast? Sacrificial rituals?

– ↑ 3 ↓ Reply ...

- PISA, because it happily trades anything in favor of faster search, so it sets a good north star in terms of search performance.

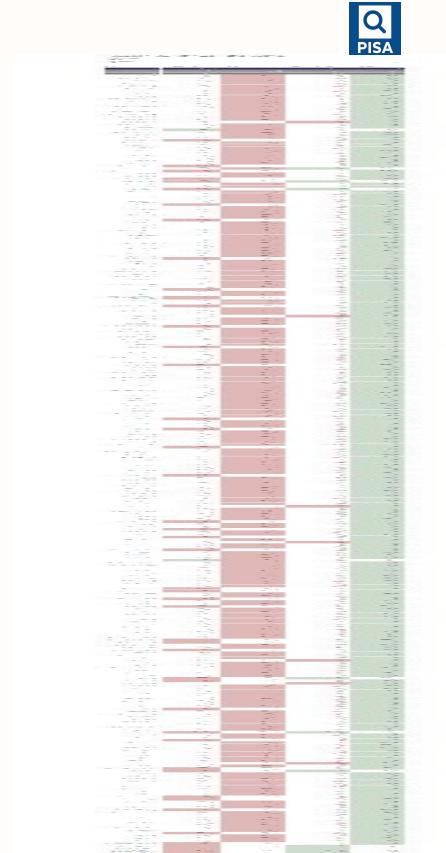
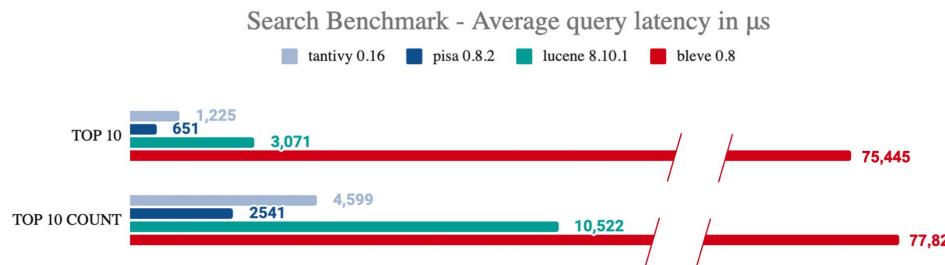
https://www.reddit.com/r/rust/comments/nu1jc7/tantivy_v015_released_now_backed_by_quickwit_inc/
<https://jpountz.github.io/2025/05/12/analysis-of-Search-Benchmark-the-Game.html>

Search Benchmark, the Game

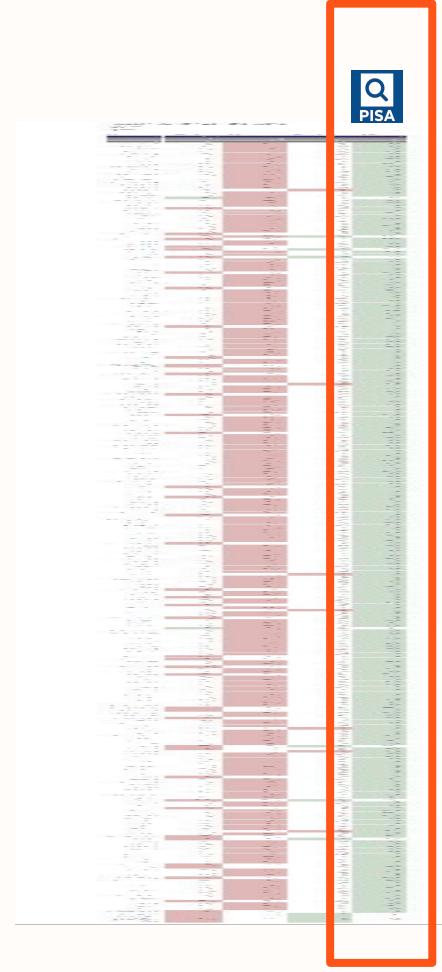
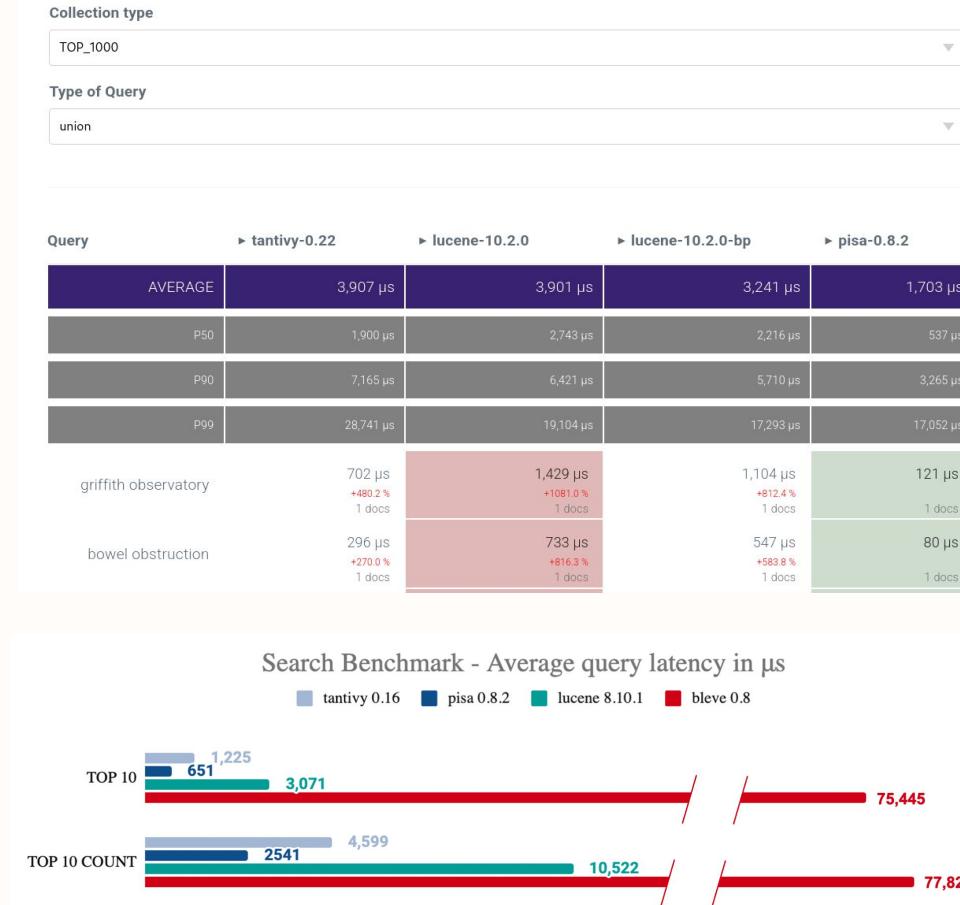
Collection type: TOP_1000

Type of Query: union

| Query | ▶ tantivy-0.22 | ▶ lucene-10.2.0 | ▶ lucene-10.2.0-bp | ▶ pisa-0.8.2 |
|----------------------|------------------------------|---------------------------------|--------------------------------|------------------|
| AVERAGE | 3,907 µs | 3,901 µs | 3,241 µs | 1,703 µs |
| P50 | 1,900 µs | 2,743 µs | 2,216 µs | 537 µs |
| P90 | 7,165 µs | 6,421 µs | 5,710 µs | 3,265 µs |
| P99 | 28,741 µs | 19,104 µs | 17,293 µs | 17,052 µs |
| griffith observatory | 702 µs +480.2 % 1 docs | 1,429 µs +1081.0 % 1 docs | 1,104 µs +812.4 % 1 docs | 121 µs 1 docs |
| bowel obstruction | 296 µs +270.0 % 1 docs | 733 µs +816.3 % 1 docs | 547 µs +583.8 % 1 docs | 80 µs 1 docs |



Search Benchmark, the Game



Session I: Indexing and Retrieval

Setup

Links and Downloads

For the practical component, we will need to grab some data, and have some instructions ready.

Prerequisite: You have your own machine with Docker installed. I will step through on my own machine if you don't have access.

You can also experiment with the tutorial at any time (later).

Tutorial: <https://shorturl.at/VExpG>

Aka: <https://github.com/pisa-engine/pisa/blob/main/test/docker/tutorial/instructions.md>

Please at least initiate the data download, and the docker image download.

Links and Downloads

Efficient In-Memory Inverted Indexes

Prerequisites

Download input data

First, [download the data package](#). It is around 1.5GB, and about 5GB once decompressed.

Move it somewhere on your machine, and unpack it. Make a note of the full path as we will need it shortly.

```
mkdir -p "$HOME/sigir25-input-data"  
mv sigir25.zip $HOME/sigir25-input-data  
cd $HOME/sigir25-input-data  
unzip sigir25.zip
```

Download container image

Next, [download the image](#). Then, load it locally:

```
docker load < pisa-tutorial.tar.gz
```

You can also use any compatible container management tool, such as `podman`.

Session I: Indexing and Retrieval

Theory

Revision: Text Indexing

Document 0

search is cool

Document 1

search is fun

Document 2

search is fun

for everyone

Revision: Text Indexing

Document 0

search is cool

Lexicon

Postings Lists

Document 1

search is fun

Document 2

search is fun
for everyone

Revision: Text Indexing

Document 0

search is cool

Lexicon

Postings Lists

Document 1

search is fun

Document 2

search is fun
for everyone

Revision: Text Indexing

Document 0

search is cool

Document 1

search is fun

Document 2

search is fun
for everyone

Lexicon

search

Postings Lists

Revision: Text Indexing

Document 0

search is cool

Document 1

search is fun

Document 2

search is fun
for everyone

Lexicon

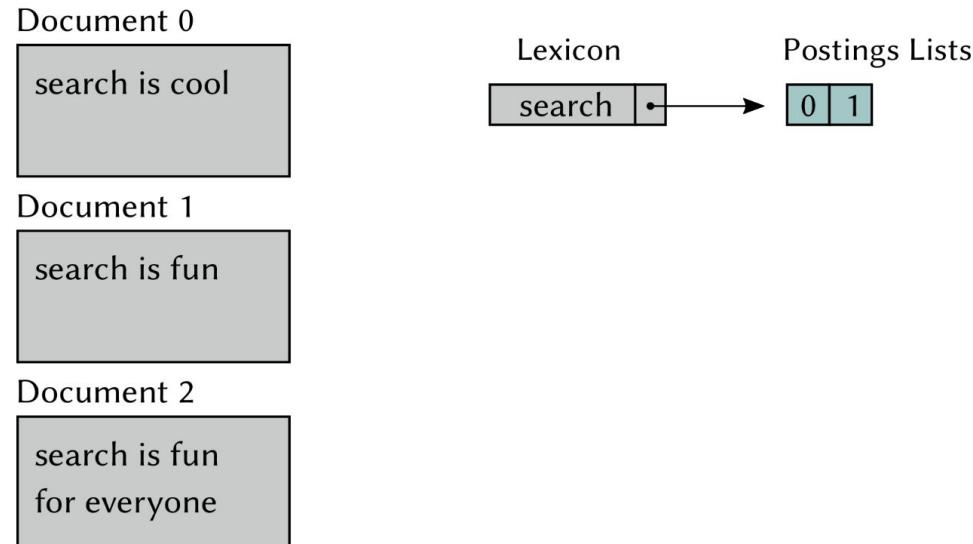
search

Postings Lists

0 1



Revision: Text Indexing



Revision: Text Indexing

Document 0

search **is**-cool

Document 1

search is fun

Document 2

search is fun
for everyone

Lexicon

search

Postings Lists

0 1



Revision: Text Indexing

Document 0

search is cool

Document 1

search is fun

Document 2

search is fun
for everyone

Lexicon

search

Postings Lists

0 1



Revision: Text Indexing

Document 0

search is cool

Document 1

search is fun

Document 2

search is fun
for everyone

Lexicon

| | | |
|--------|---|--|
| search | • | |
| cool | | |

Postings Lists

| | |
|---|---|
| 0 | 1 |
|---|---|

Revision: Text Indexing

Document 0

search is cool

Document 1

search is fun

Document 2

search is fun

for everyone

Lexicon

| | |
|--------|---|
| search | • |
| cool | • |

Postings Lists

| | |
|---|---|
| 0 | 1 |
| 0 | 1 |

Revision: Text Indexing

Document 0

search is cool

Document 1

search is fun

Document 2

search is fun

for everyone

Lexicon

| | |
|--------|---|
| search | • |
| cool | • |

Postings Lists

| | |
|---|---|
| 0 | 1 |
| 0 | 1 |

Revision: Text Indexing

Document 0

search is cool

Document 1

search is fun

Document 2

search is fun
for everyone

Lexicon

| | |
|--------|---|
| search | • |
| cool | • |

Postings Lists

| | |
|---|---|
| 0 | 1 |
| 0 | 1 |

Lexicon

| | |
|--------|---|
| search | • |
| cool | • |

Postings Lists

| | |
|---|---|
| 0 | 1 |
| 0 | 1 |

Revision: Text Indexing

Document 0

search is cool

Document 1

search is fun

Document 2

search is fun

for everyone

Lexicon

| | |
|--------|---|
| search | • |
| cool | • |

Postings Lists

| | |
|---|---|
| 0 | 1 |
| 0 | 1 |

Revision: Text Indexing

Document 0

search is cool

Document 1

search is fun

Document 2

search is fun

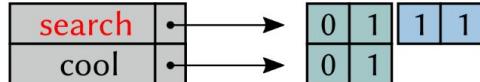
for everyone

Lexicon

| | |
|--------|---|
| search | • |
| cool | • |

Postings Lists

| | | | |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 0 | 1 | | |



Revision: Text Indexing

Document 0

search is cool

Document 1

search is fun

Document 2

search is fun
for everyone

Lexicon

| | |
|--------|---|
| search | • |
| cool | • |

Postings Lists

| | | | |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 0 | 1 | | |

...

Revision: Text Indexing

Document 0

search is cool

Document 1

search is fun

Document 2

search is fun
for everyone

Lexicon

| | | |
|----------|---|---|
| search | • | → |
| cool | • | → |
| fun | • | → |
| everyone | • | → |

Postings Lists

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 2 | 1 |
| 0 | 1 | | | | |
| 1 | 1 | | | 2 | 1 |
| 2 | 1 | | | | |

Revision: Text Indexing

Document 0

search is cool

Document 1

search is fun

Document 2

search is fun
for everyone

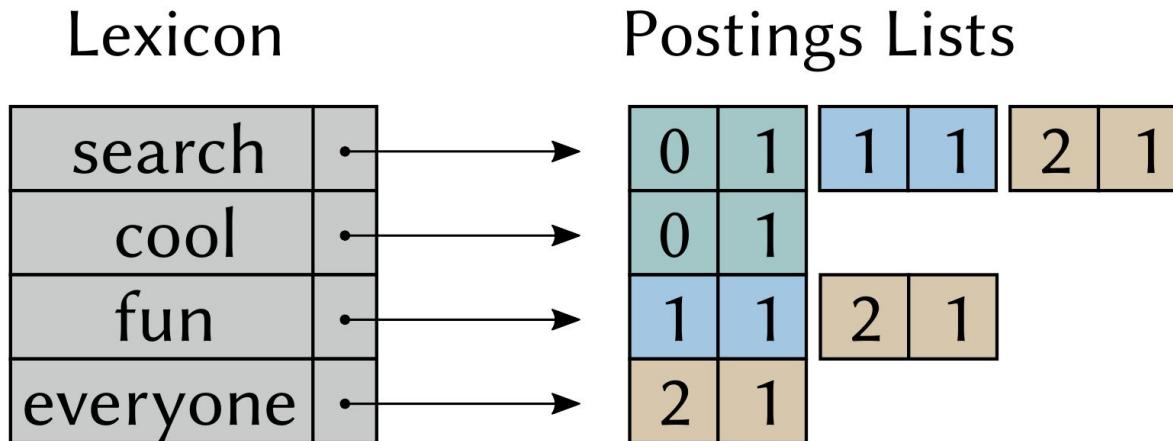
Lexicon

| | |
|----------|---|
| search | • |
| cool | • |
| fun | • |
| everyone | • |

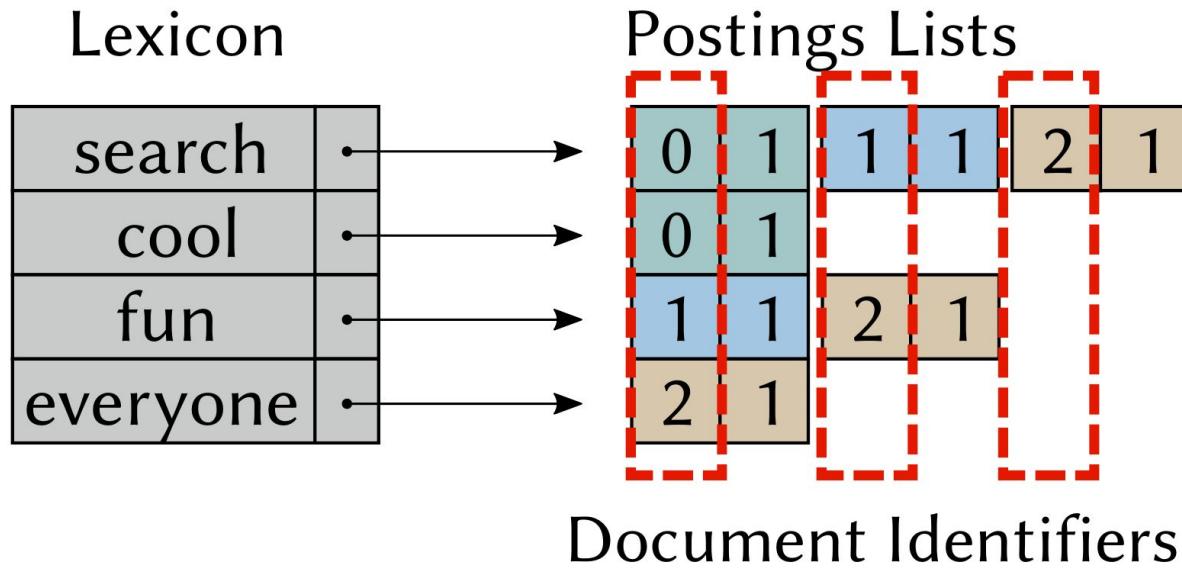
Postings Lists

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 2 | 1 |
| 0 | 1 | | | | |
| 1 | 1 | | | 2 | 1 |
| 2 | 1 | | | | |

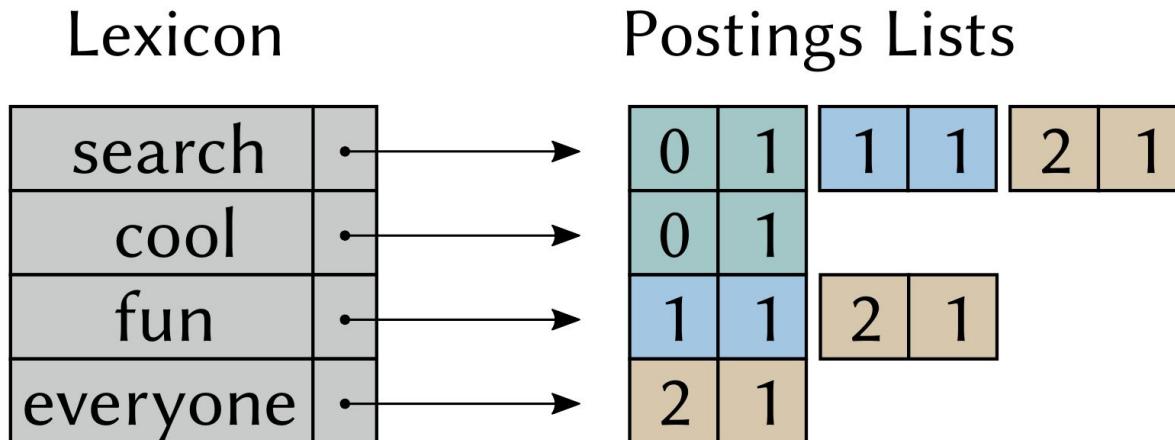
Revision: Inverted Indexes



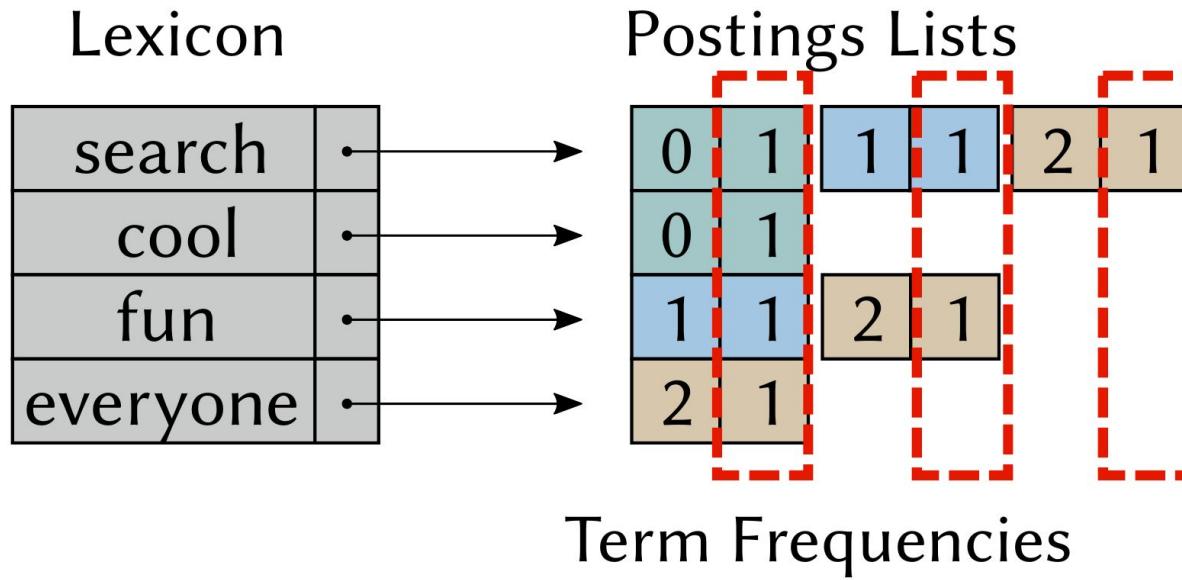
Revision: Inverted Indexes



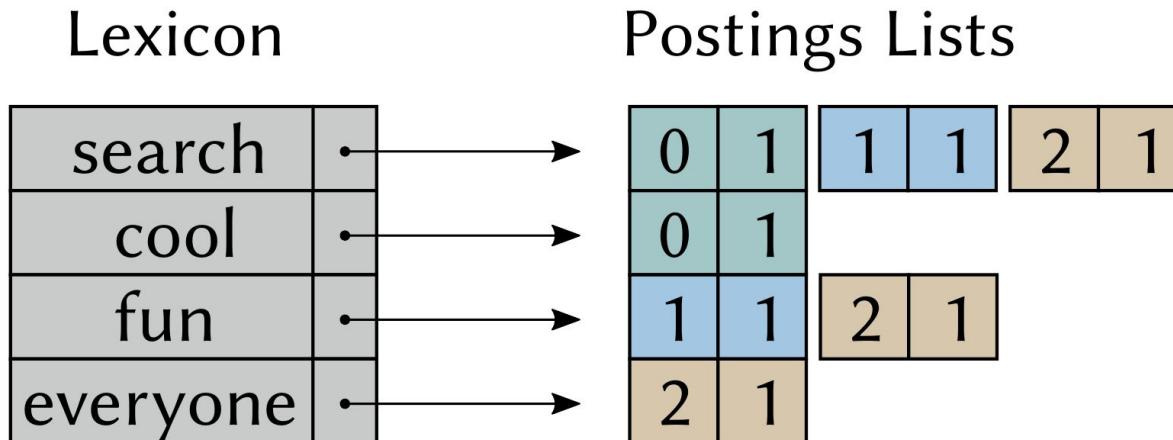
Revision: Inverted Indexes



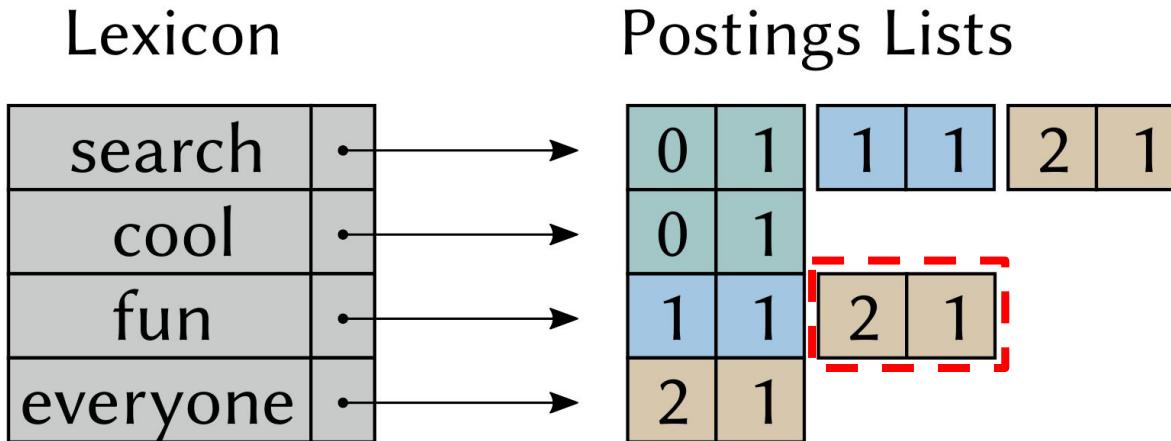
Revision: Inverted Indexes



Revision: Inverted Indexes

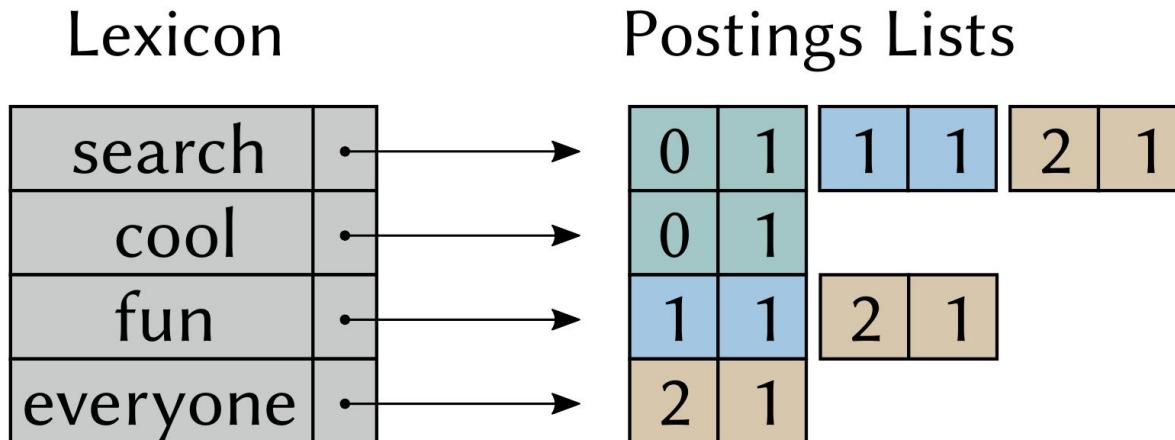


Revision: Inverted Indexes

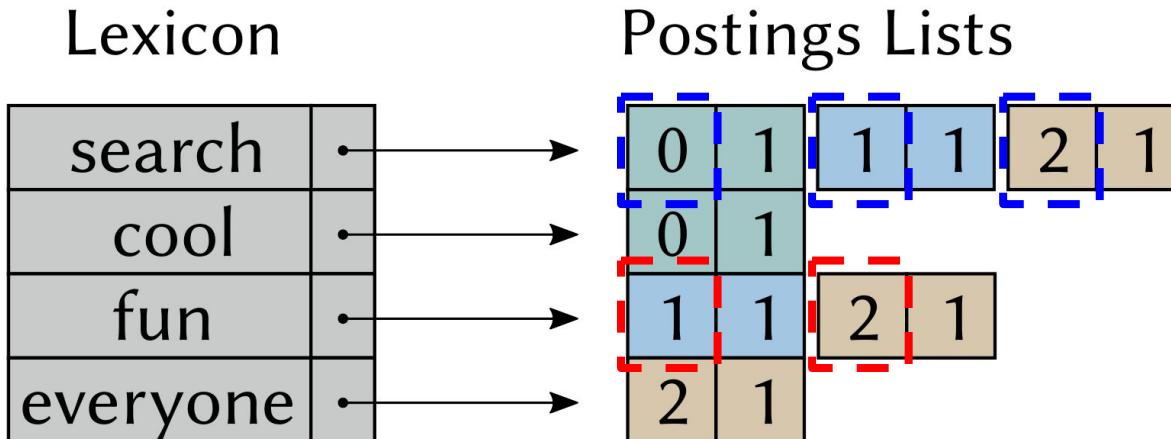


A single posting is a **document identifier** and **term frequency pair**. Document 2 contains the word “fun” once.

Revision: Inverted Indexes



Revision: Inverted Indexes



Within a given postings list,
document identifiers are
strictly increasing.

Compressed Postings

Storing Postings in Practice

DocIDs

| | | | | | | | | |
|----|----|----|----|----|----|----|----|-----|
| 12 | 14 | 27 | 29 | 30 | 55 | 59 | 86 | ... |
|----|----|----|----|----|----|----|----|-----|

Frequencies

| | | | | | | | | |
|---|---|---|---|---|---|---|---|-----|
| 3 | 4 | 3 | 1 | 1 | 5 | 2 | 3 | ... |
|---|---|---|---|---|---|---|---|-----|

Compressed Postings

Storing Postings in Practice

DocIDs

| | | | | | | | | |
|----|----|----|----|----|----|----|----|-----|
| 12 | 14 | 27 | 29 | 30 | 55 | 59 | 86 | ... |
|----|----|----|----|----|----|----|----|-----|

Frequencies

| | | | | | | | | |
|---|---|---|---|---|---|---|---|-----|
| 3 | 4 | 3 | 1 | 1 | 5 | 2 | 3 | ... |
|---|---|---|---|---|---|---|---|-----|

Compressed Postings

Storing Postings in Practice

DocIDs

| | | | | | | | | |
|----|----|----|----|----|----|----|----|-----|
| 12 | 14 | 27 | 29 | 30 | 55 | 59 | 86 | ... |
|----|----|----|----|----|----|----|----|-----|

 $14 - 12 = 2$

Frequencies

| | | | | | | | | |
|---|---|---|---|---|---|---|---|-----|
| 3 | 4 | 3 | 1 | 1 | 5 | 2 | 3 | ... |
|---|---|---|---|---|---|---|---|-----|

Compressed Postings

Storing Postings in Practice

DocIDs

| | | | | | | | | |
|----|---|----|----|----|----|----|----|-----|
| 12 | 2 | 27 | 29 | 30 | 55 | 59 | 86 | ... |
|----|---|----|----|----|----|----|----|-----|

 $14 - 12 = 2$

Frequencies

| | | | | | | | | |
|---|---|---|---|---|---|---|---|-----|
| 3 | 4 | 3 | 1 | 1 | 5 | 2 | 3 | ... |
|---|---|---|---|---|---|---|---|-----|

Compressed Postings

Storing Postings in Practice

| | | | | | | | | | |
|--------|----|---|----|----|----|----|----|----|-----|
| DocIDs | 12 | 2 | 27 | 29 | 30 | 55 | 59 | 86 | ... |
| | 14 | | | | | | | | |

| | | | | | | | | | |
|-------------|---|---|---|---|---|---|---|---|-----|
| Frequencies | 3 | 4 | 3 | 1 | 1 | 5 | 2 | 3 | ... |
|-------------|---|---|---|---|---|---|---|---|-----|

Compressed Postings

Storing Postings in Practice

| | | | | | | | | | |
|--------|----|---|----|----|----|----|----|----|-----|
| DocIDs | 12 | 2 | 27 | 29 | 30 | 55 | 59 | 86 | ... |
| | | | 14 | | | | | | |

| | | | | | | | | | |
|-------------|---|---|---|---|---|---|---|---|-----|
| Frequencies | 3 | 4 | 3 | 1 | 1 | 5 | 2 | 3 | ... |
|-------------|---|---|---|---|---|---|---|---|-----|

Compressed Postings

Storing Postings in Practice

| | | | | | | | | | |
|----------------|----|---|----|----|----|----|----|----|-----|
| DocIDs | 12 | 2 | 27 | 29 | 30 | 55 | 59 | 86 | ... |
| $27 - 14 = 13$ | | | | | | | | | |

| | | | | | | | | | |
|-------------|---|---|---|---|---|---|---|---|-----|
| Frequencies | 3 | 4 | 3 | 1 | 1 | 5 | 2 | 3 | ... |
|-------------|---|---|---|---|---|---|---|---|-----|

Compressed Postings

Storing Postings in Practice

DocIDs  ...

Frequencies  ...

Compressed Postings

Storing Postings in Practice

DocIDs

| | | | | | | | |
|----|---|----|---|---|----|---|----|
| 12 | 2 | 13 | 2 | 2 | 25 | 4 | 27 |
|----|---|----|---|---|----|---|----|

...

Frequencies

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 4 | 3 | 1 | 1 | 5 | 2 | 3 |
|---|---|---|---|---|---|---|---|

...

This is called **delta coding** and is a common pre-processing step used to make integer compression codecs better.

While out of scope for today, integer codecs are typically **more effective for smaller integers**.

Compressed Postings

Storing Postings in Practice

DocIDs  ...

Frequencies  ...

Compressed Postings

Storing Postings in Practice



Prefix-Sum Problem

Storing Postings in Practice



Prefix-Sum Problem

Storing Postings in Practice

DocIDs



Frequencies



But what if I want to access the
 n th document identifier?



Prefix-Sum Problem

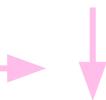
Storing Postings in Practice

Sum these deltas up

DocIDs



But what if I want to access the n th document identifier?



Frequencies



Prefix-Sum Problem

Storing Postings in Practice



But what if I want to access the n th document identifier?

Prefix-Sum Problem

Storing Postings in Practice

But what if I want to access the n th document identifier?



Break into fixed-sized blocks



Prefix-Sum Problem

Storing Postings in Practice

But what if I want to access the n th document identifier?



Break into fixed-sized blocks

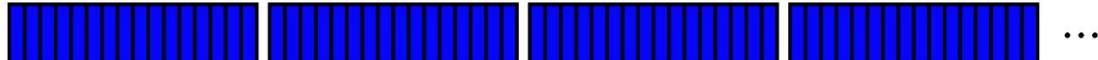


Store the *first* identifier of
each *block* directly - then
prefix sums are only required
within each block!

Prefix-Sum Problem

Storing Postings in Practice

DocIDs

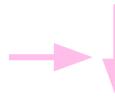


Break into fixed-sized blocks

Frequencies



But what if I want to access the n th document identifier?



Store the *first* identifier of each *block* directly - then prefix sums are only required within each block!

Block-Based Indexes

Storing Postings in Practice



Break into fixed-sized blocks



Block-Based Indexes

Storing Postings in Practice



Compress each block individually



Block-Based Indexes

Storing Postings in Practice



Interleave for better memory locality



Block-Based Indexes

Storing Postings in Practice



Interleave for better memory locality



In-memory (or on-disk) postings

Postings List Compression

Table 1. Timeline of techniques.

| | | | |
|------|--|------|---|
| 1949 | Shannon-Fano [32, 93] | 2005 | Simple-9, Relative-10, and Carryover-12 [3]; RBUC [60] |
| 1952 | Huffman [43] | 2006 | PForDelta [114]; BASC [61] |
| 1963 | Arithmetic [1] ¹ | 2008 | Simple-16 [112]; Tournament [100] |
| 1966 | Golomb [40] | 2009 | ANS [27]; Varint-GB [23]; Opt-PFor [111] |
| 1971 | Elias-Fano [30, 33]; Rice [87] | 2010 | Simple8b [4]; VSE [96]; SIMD-Gamma [91] |
| 1972 | Variable-Byte and Nibble [101] | 2011 | Varint-G8IU [97]; Parallel-PFor [5] |
| 1975 | Gamma and Delta [31] | 2013 | DAC [12]; Quasi-Succinct [107] |
| 1978 | Exponential Golomb [99] | 2014 | partitioned Elias-Fano [73]; QMX [103]; Roaring [15, 51, 53] |
| 1985 | Fibonacci-based [6, 37] | 2015 | BP32, SIMD-BP128, and SIMD-FastPFor [50]; Masked-VByte [84] |
| 1986 | Hierarchical bit-vectors [35] | 2017 | clustered Elias-Fano [80] |
| 1988 | Based on Front Coding [16] | | Stream-VByte [52]; ANS-based [63, 64]; |
| 1996 | Interpolative [65, 66] | | Opt-VByte [83]; SIMD-Delta [104]; general-purpose compression libraries [77]; |
| 1998 | Frame-of-Reference (For) [39]; modified Rice [2] | | |
| 2003 | SC-dense [11] | | |
| 2004 | Zeta [8, 9] | | |
| | | 2019 | DINT [79]; Slicing [78] |

Postings List Compression

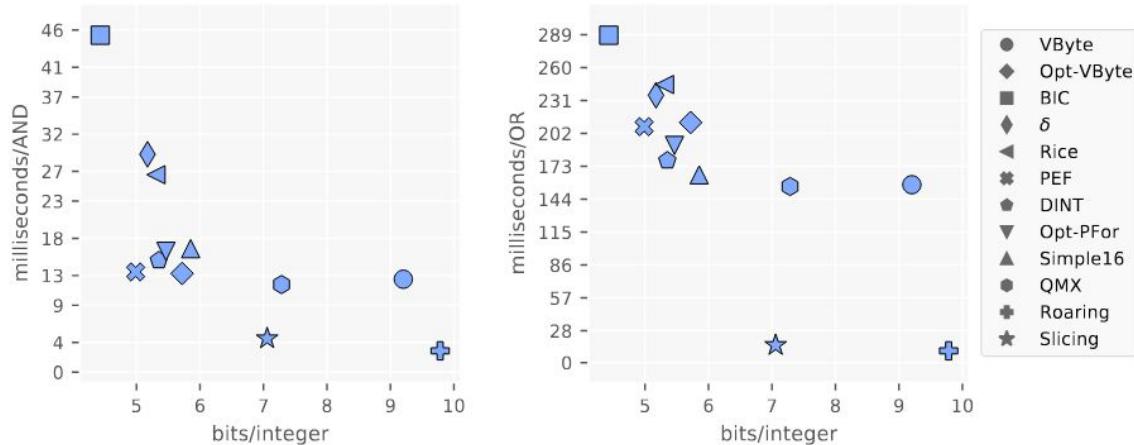


Fig. 7. Space/time trade-off curves for the ClueWeb09 dataset.

Key message: It is almost always a trade-off. Smaller codecs take longer to encode/decode, but save space.

The “right” choice depends on where you wish to operate on the Pareto frontier.

Revision: Basic Querying

Now we have our inverted index, **how can we query it?**

- **Lots** of flavours of query...
- Let's start by revising the simple Boolean *conjunction*
- We'll then move on to ranked disjunctions, also known as top- k retrieval.
- Important to note that *matching semantics* are separate from document *ranking*.
 - That is, we can decide to only match documents containing **all** query terms, but we might also decide to rank them on the way through! This would be a *ranked conjunction*.

Revision: Boolean Conjunctions

best

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 12 | 14 | 27 | 29 | 30 | 55 | 59 | 86 |
|----|----|----|----|----|----|----|----|

coffee

| | | | | |
|---|----|----|----|----|
| 8 | 11 | 12 | 55 | 74 |
|---|----|----|----|----|

melbourne

| | | | | | |
|---|----|----|----|----|----|
| 2 | 12 | 30 | 45 | 51 | 55 |
|---|----|----|----|----|----|

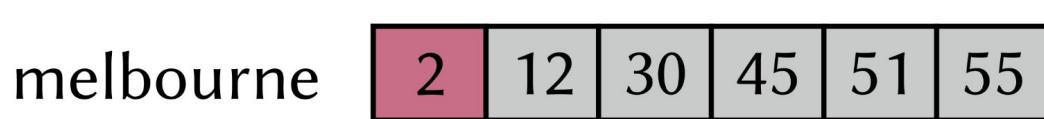
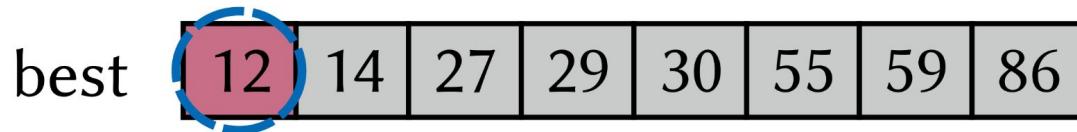
Revision: Boolean Conjunctions

best  12 14 27 29 30 55 59 86

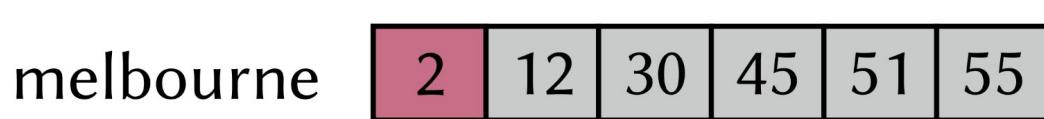
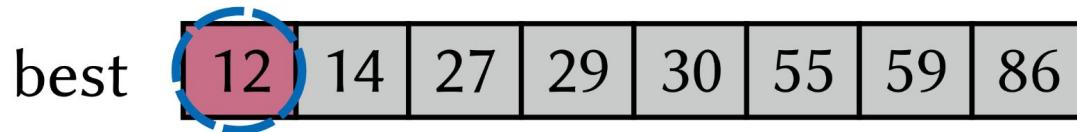
coffee  8 11 12 55 74

melbourne  2 12 30 45 51 55

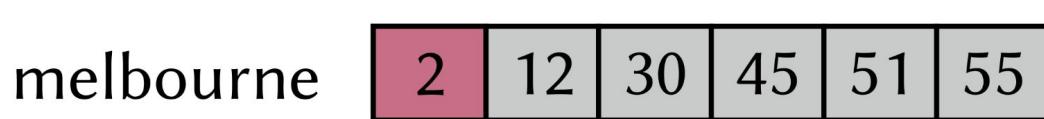
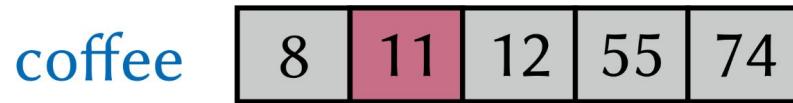
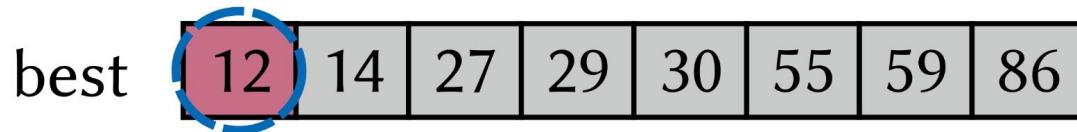
Revision: Boolean Conjunctions



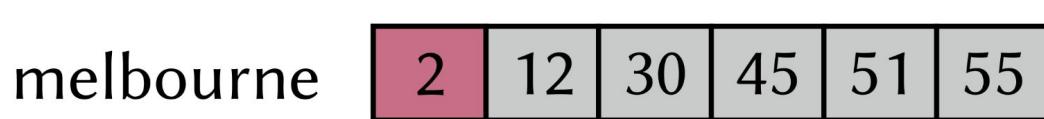
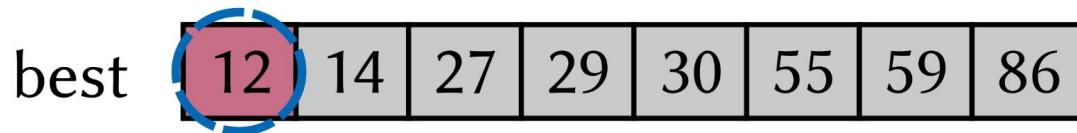
Revision: Boolean Conjunctions



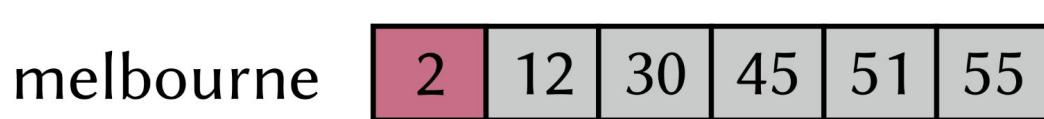
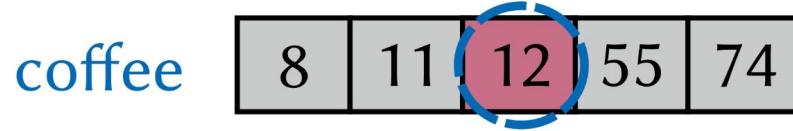
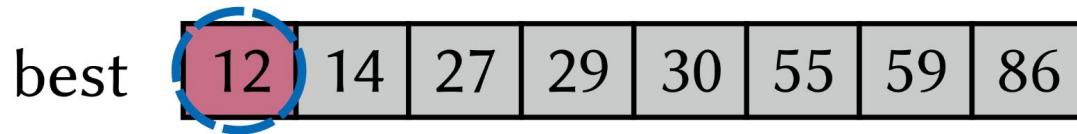
Revision: Boolean Conjunctions



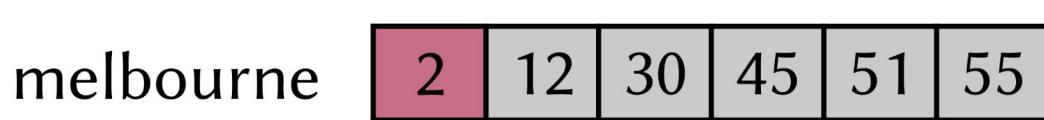
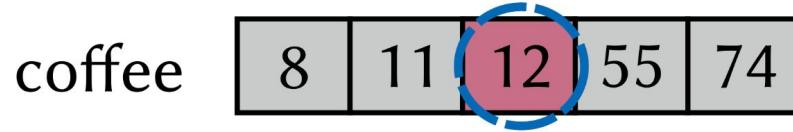
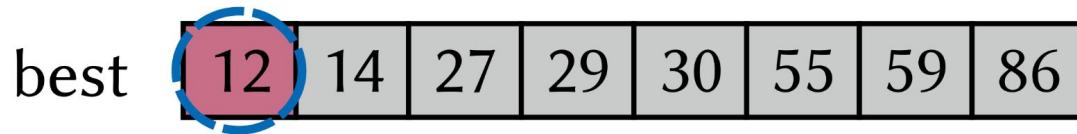
Revision: Boolean Conjunctions



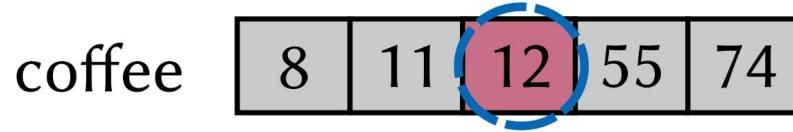
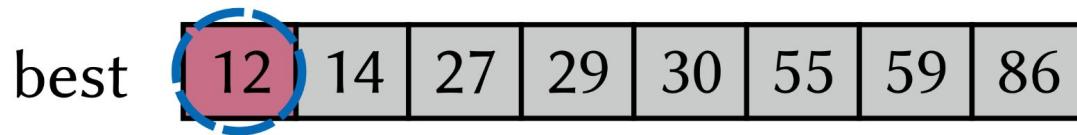
Revision: Boolean Conjunctions



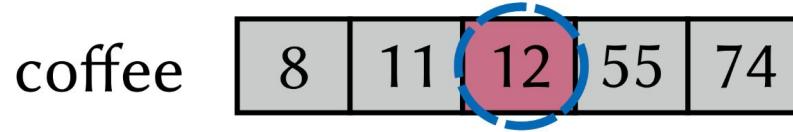
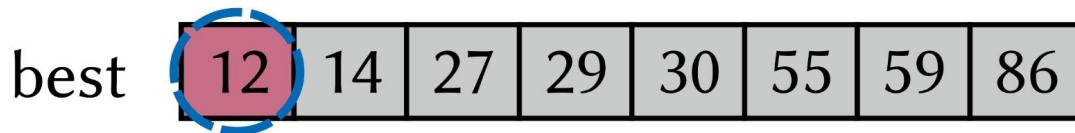
Revision: Boolean Conjunctions



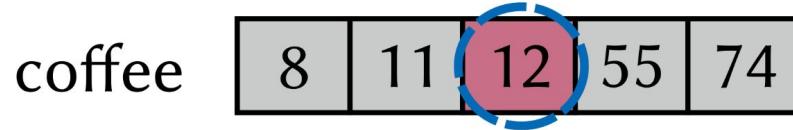
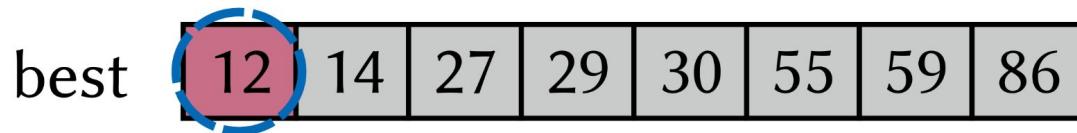
Revision: Boolean Conjunctions



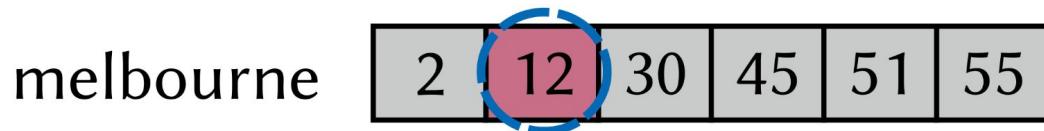
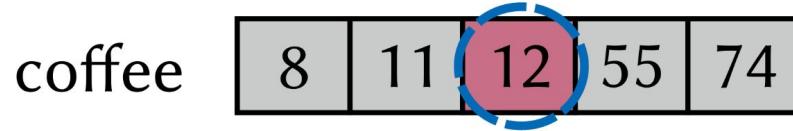
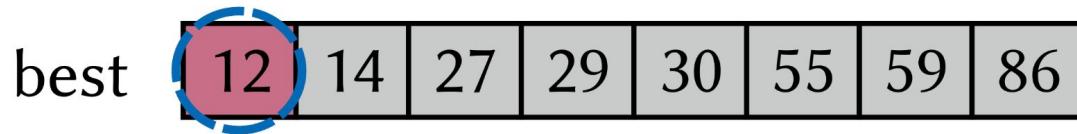
Revision: Boolean Conjunctions



Revision: Boolean Conjunctions

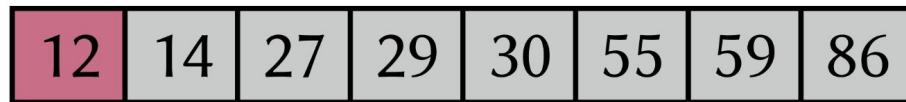


Revision: Boolean Conjunctions



Revision: Boolean Conjunctions

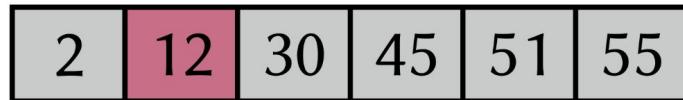
best



coffee

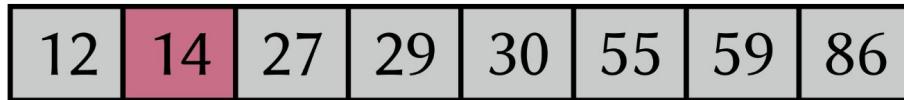


melbourne



Revision: Boolean Conjunctions

best



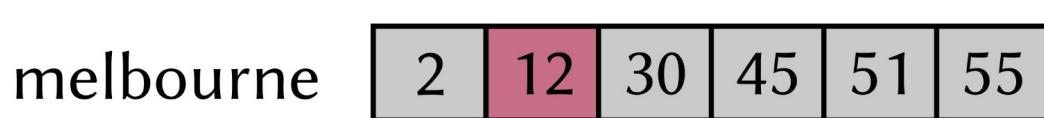
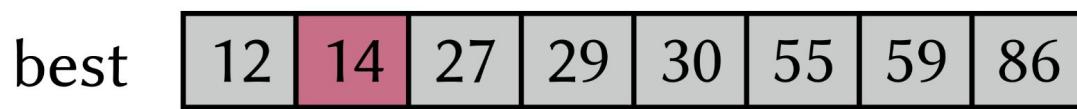
coffee



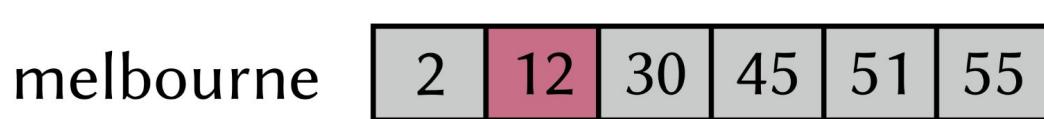
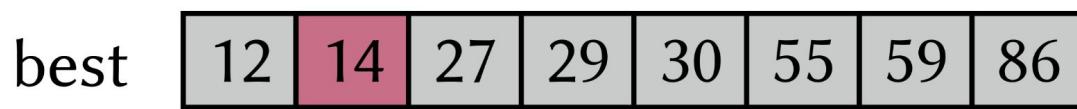
melbourne



Revision: Boolean Conjunctions

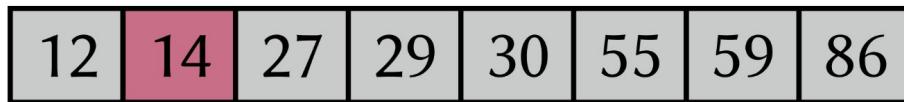


Revision: Boolean Conjunctions



Revision: Boolean Conjunctions

best



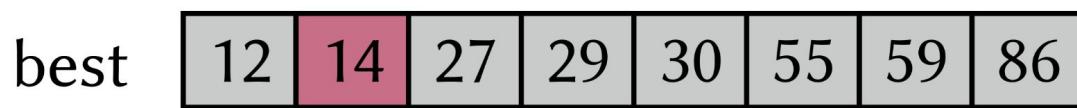
coffee



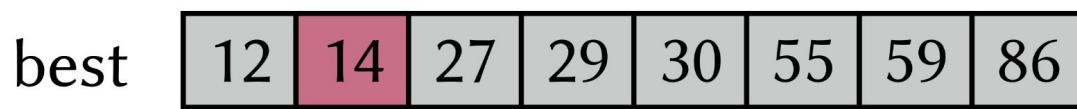
melbourne



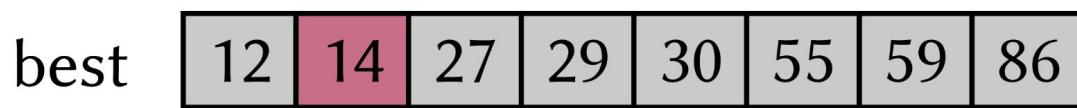
Revision: Boolean Conjunctions



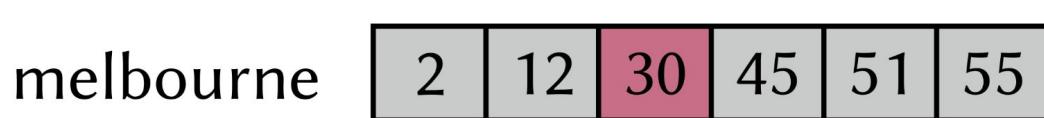
Revision: Boolean Conjunctions



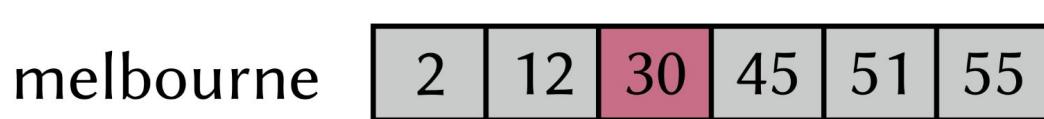
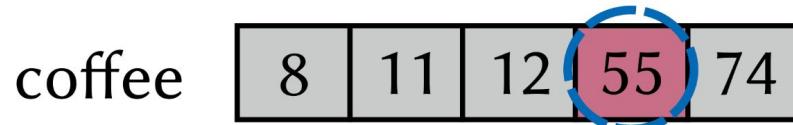
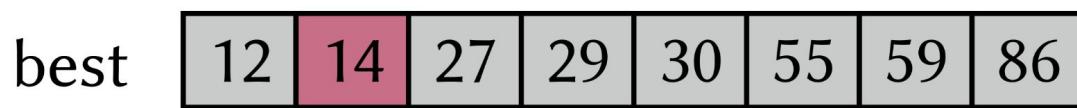
Revision: Boolean Conjunctions



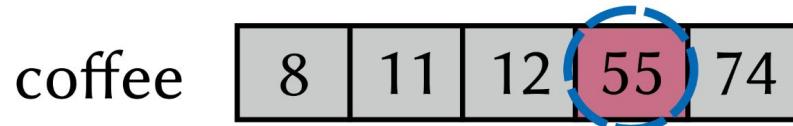
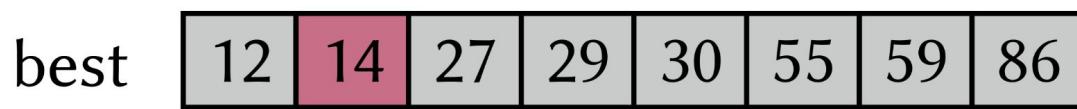
Revision: Boolean Conjunctions



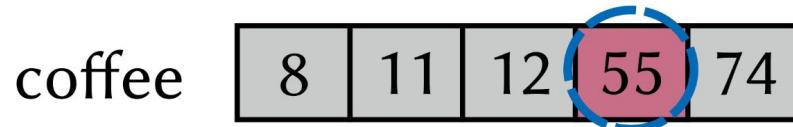
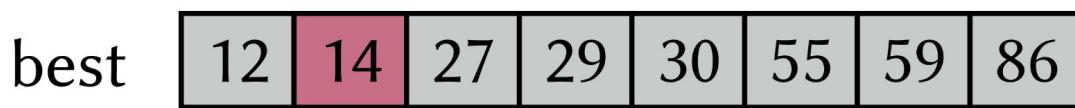
Revision: Boolean Conjunctions



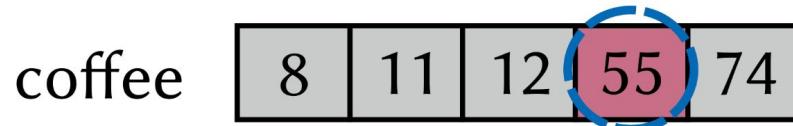
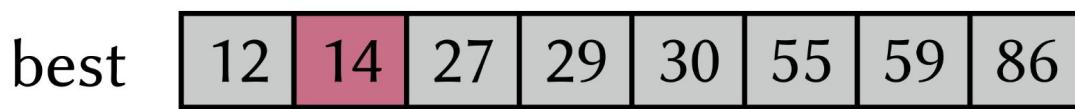
Revision: Boolean Conjunctions



Revision: Boolean Conjunctions

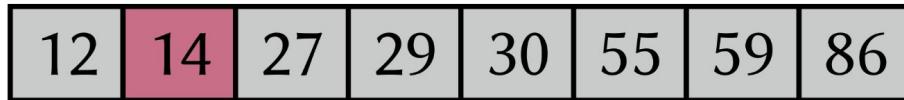


Revision: Boolean Conjunctions

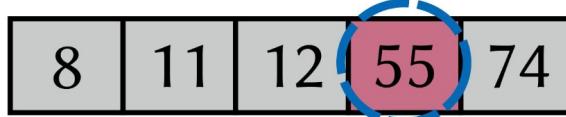


Revision: Boolean Conjunctions

best



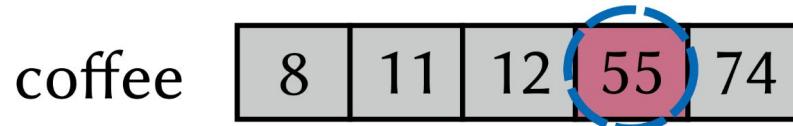
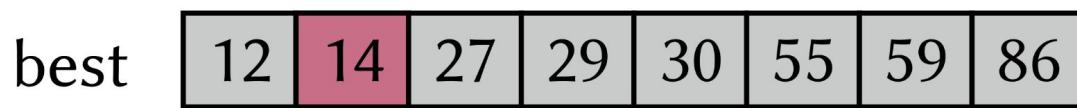
coffee



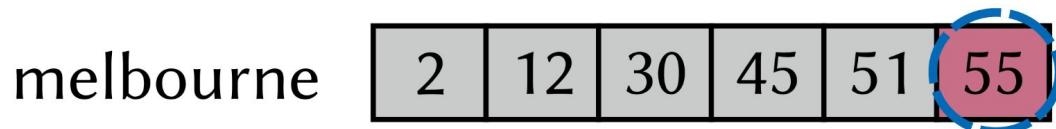
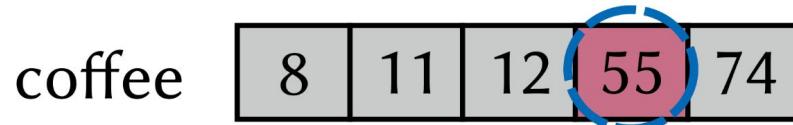
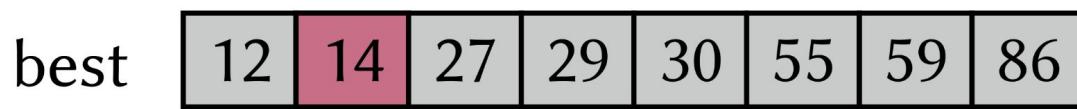
melbourne



Revision: Boolean Conjunctions

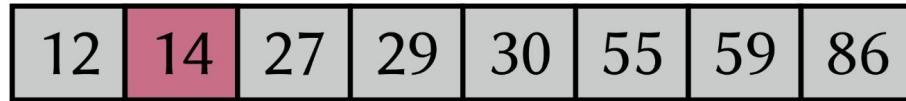


Revision: Boolean Conjunctions

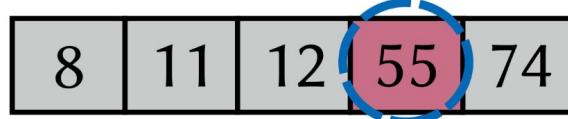


Revision: Boolean Conjunctions

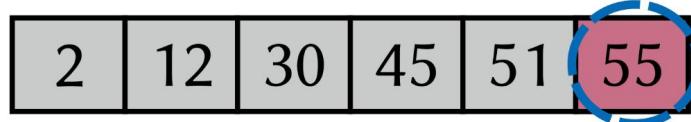
best



coffee



melbourne

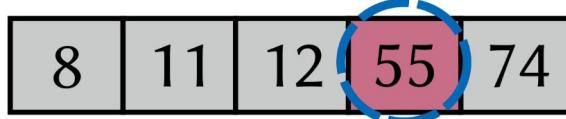


Revision: Boolean Conjunctions

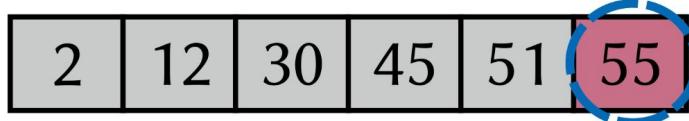
best



coffee

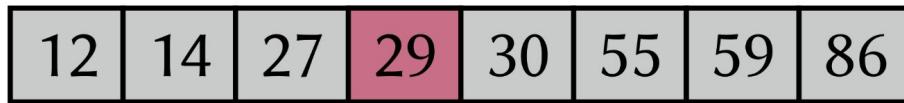


melbourne

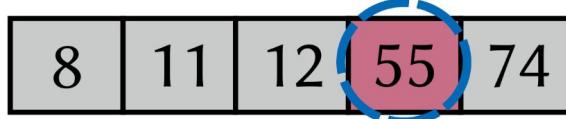


Revision: Boolean Conjunctions

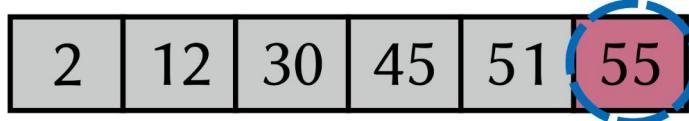
best



coffee



melbourne

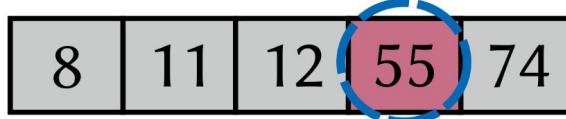


Revision: Boolean Conjunctions

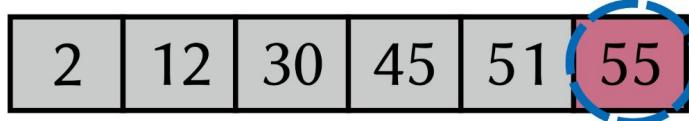
best



coffee

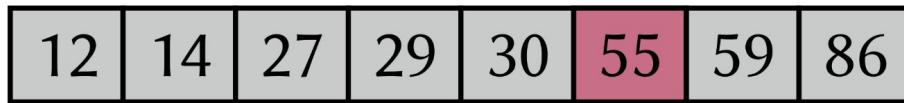


melbourne

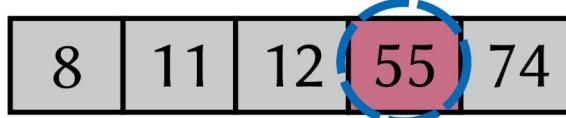


Revision: Boolean Conjunctions

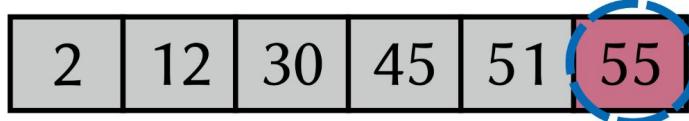
best



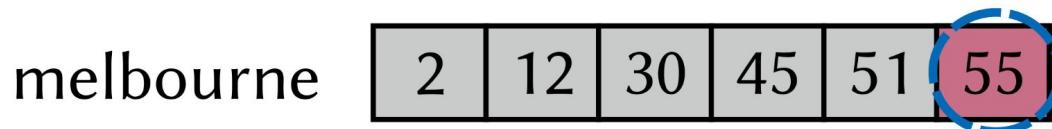
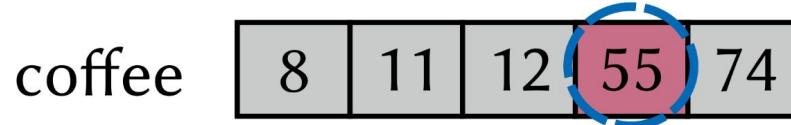
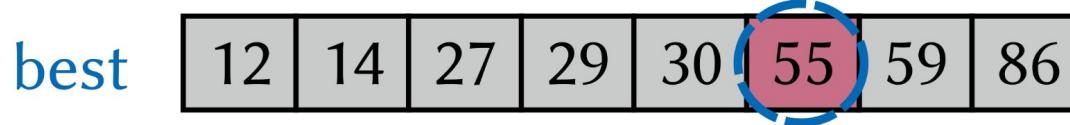
coffee



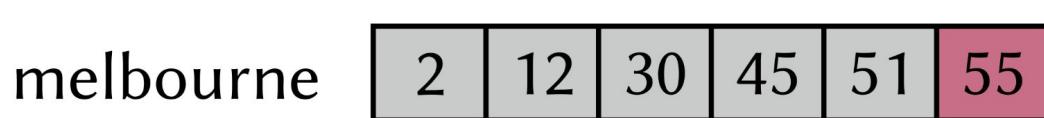
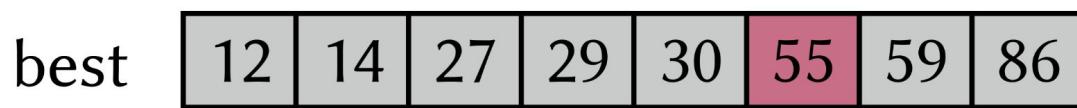
melbourne



Revision: Boolean Conjunctions



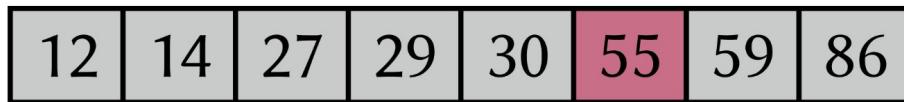
Revision: Boolean Conjunctions



Revision: Boolean Conjunctions

And so on until we run out of postings...

best



coffee

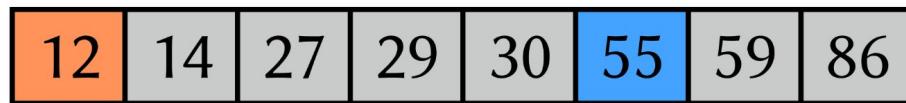


melbourne



Revision: Boolean Conjunctions

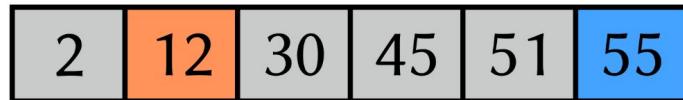
best



coffee



melbourne



Hmmm... What about ranking?

Consider our old, faithful friend, BM25 [ATIRE variant]:

$$\sum_{t \in q} \log \left(\frac{N}{df_t} \right) \cdot \frac{(k_1 + 1) \cdot tf_{td}}{k_1 \cdot \left(1 - b + b \cdot \left(\frac{L_d}{L_{avg}} \right) \right) + tf_{td}}$$

Hmmm... What about ranking?

Consider our old, faithful friend, BM25 [ATIRE variant]:

For each query term...

$$\sum_{t \in q} \log \left(\frac{N}{df_t} \right) \cdot \frac{(k_1 + 1) \cdot tf_{td}}{k_1 \cdot \left(1 - b + b \cdot \left(\frac{L_d}{L_{avg}} \right) \right) + tf_{td}}$$

Number of documents in the collection
Number of docs containing term t

Number of times t appears in “this” document
Length of “this” document / Average document length

Hmmm... What about ranking?

Consider our old, faithful friend, BM25 [ATIRE variant]:

$$\sum_{t \in q} \log \left(\frac{N}{df_t} \right) \cdot \frac{(k_1 + 1) \cdot tf_{td}}{k_1 \cdot \left(1 - b + b \cdot \left(\frac{L_d}{L_{avg}} \right) \right) + tf_{td}}$$

Hmmm... What about ranking?

Where is this stuff stored?

For each query term...

$$\sum_{t \in q} \log \left(\frac{N}{df_t} \right) \cdot \frac{(k_1 + 1) \cdot tf_{td}}{k_1 \cdot \left(1 - b + b \cdot \left(\frac{L_d}{L_{avg}} \right) \right) + tf_{td}}$$

Number of documents in the collection
Number of docs containing term t
Number of times t appears in “this” document
Length of “this” document / Average document length

Hmmm... What about ranking?

Where is this stuff stored?

For each query term...

$$\sum_{t \in q} \log \left(\frac{N}{df_t} \right) \cdot \frac{(k_1 + 1) \cdot tf_{td}}{k_1 \cdot \left(1 - b + b \cdot \left(\frac{L_d}{L_{avg}} \right) \right) + tf_{td}}$$

Number of documents in the collection

Number of times t appears in “this” document

Number of docs containing term t

Length of “this” document / Average document length

| Lexicon | Postings Lists |
|----------|----------------|
| search | 0 1 |
| cool | 0 1 |
| fun | 1 1 |
| everyone | 2 1 |

+ Array with (normalized) document lengths

Hmmm... What about ranking?

Consider our old, faithful friend, BM25 [ATIRE variant]:

$$\sum_{t \in q} \log \left(\frac{N}{df_t} \right) \cdot \frac{(k_1 + 1) \cdot tf_{td}}{k_1 \cdot \left(1 - b + b \cdot \left(\frac{L_d}{L_{avg}} \right) \right) + tf_{td}}$$

Note how the “query dependent” aspect is just which terms get used...

Hmmm... What about ranking?

Consider our old, faithful friend, BM25 [ATIRE variant]:

$$\sum_{t \in q} \left[\log \left(\frac{N}{df_t} \right) \cdot \frac{(k_1 + 1) \cdot tf_{td}}{k_1 \cdot \left(1 - b + b \cdot \left(\frac{L_d}{L_{avg}} \right) \right) + tf_{td}} \right]$$

Note how the “query dependent” aspect is just which terms get used...

So, we can pre-compute the **document/term impacts!**

Hmmm... What about ranking?

Consider our old, faithful friend, BM25 [ATIRE variant]:

$$\sum_{t \in q} I_{d,t}$$

Note how the “query dependent” aspect is just which terms get used...

So, we can pre-compute the **document/term impacts!**

Hmmm... What about ranking?

Consider our old, faithful friend, BM25 [ATIRE variant]:

$$\sum_{t \in q} I_{d,t}$$

| Lexicon | | Postings Lists | | | |
|----------|---|----------------|---|---|---|
| search | . | 0 | 1 | 1 | 1 |
| cool | . | 0 | 1 | | |
| fun | . | 1 | 1 | 2 | 1 |
| everyone | . | 2 | 1 | | |

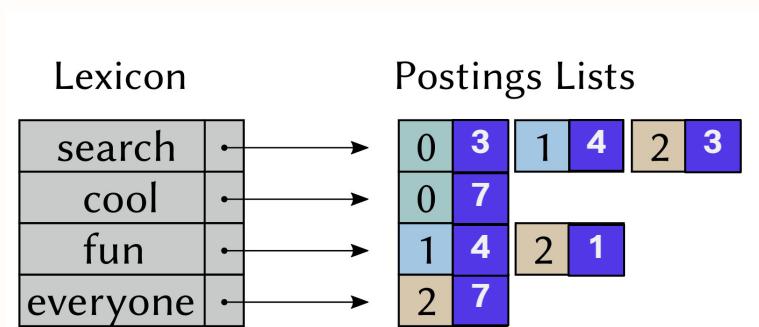
Note how the “query dependent” aspect is just which terms get used...

So, we can pre-compute the **document/term impacts!** And **store them directly in the index!**

Hmmm... What about ranking?

Quantized Scoring!

$$\sum_{t \in q} I_{d,t}$$



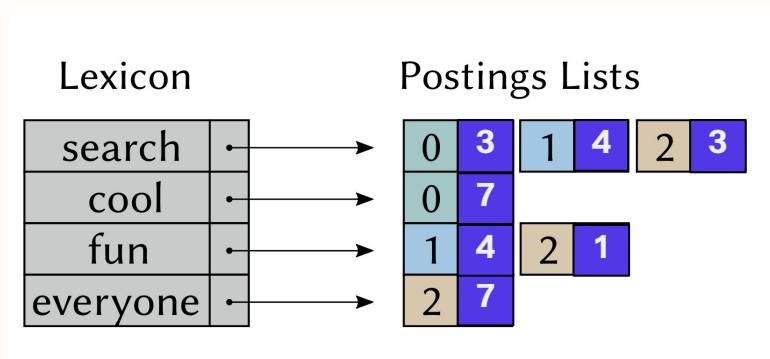
Note how the “query dependent” aspect is just which terms get used...

So, we can pre-compute the **document/term impacts!** And **store them directly in the index!** [After global normalization]

Quantized Scorers

For the remainder of this tutorial, we will assume “sum of impact” scoring

$$\sum_{t \in q} I_{d,t}$$



Instead of storing term frequencies, our postings lists will store quantized impacts.

Top- k Retrieval

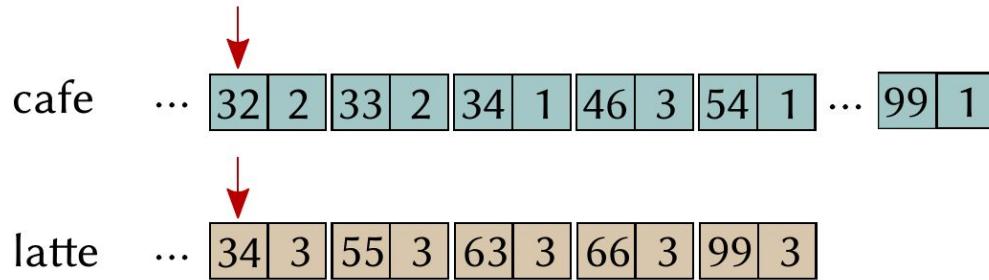
Instead of relying on conjunctive matching, let's trust our ranking function and allow *disjunctive* matching.

Naive algorithm: Scan across all postings lists maintaining a min-heap of the k best “so far” documents. For each document, compute its score, and add it to the heap if it *beats* the current top element.

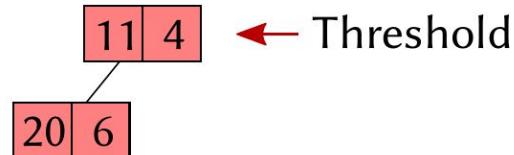
At the end of traversal, the heap contains the top- k docs.

Top- k Retrieval

Term Postings Lists

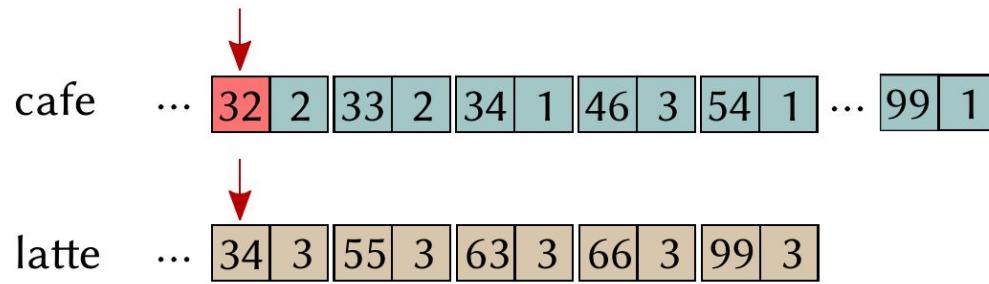


Top- k documents (min-heap) ($k = 2$)

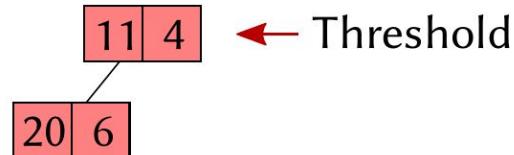


Top- k Retrieval

Term Postings Lists

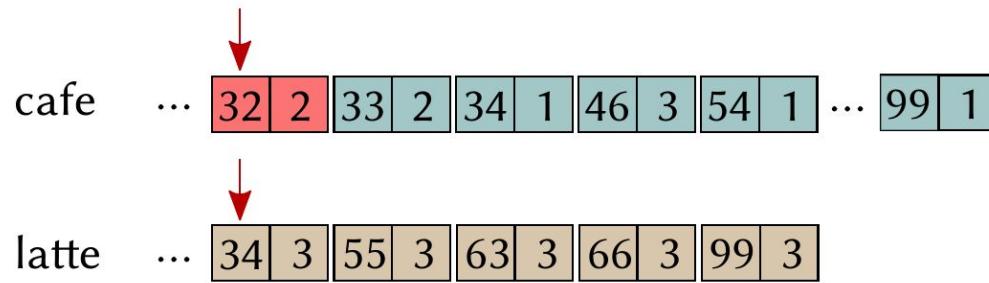


Top- k documents (min-heap) ($k = 2$)

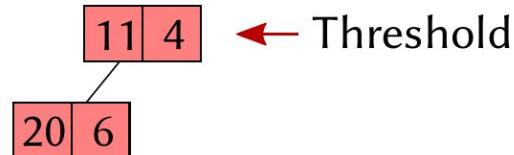


Top- k Retrieval

Term Postings Lists

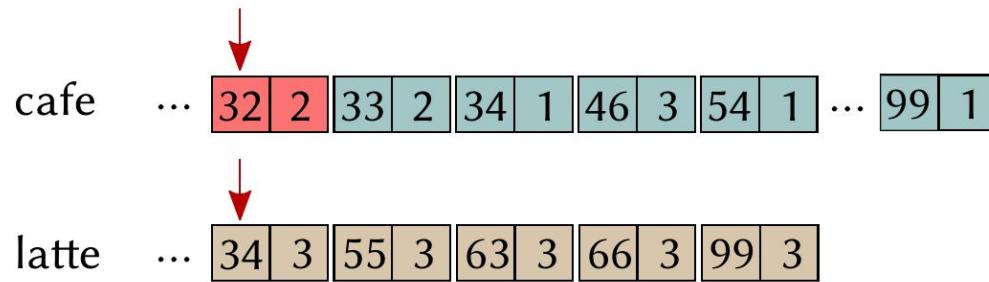


Top- k documents (min-heap) ($k = 2$)

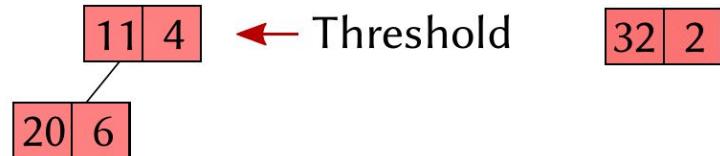


Top- k Retrieval

Term Postings Lists

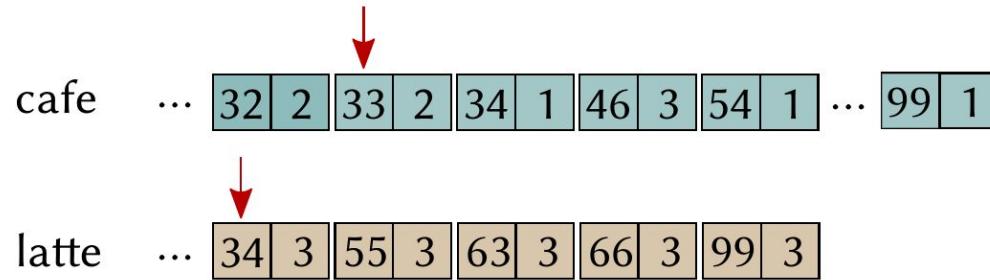


Top- k documents (min-heap) ($k = 2$)

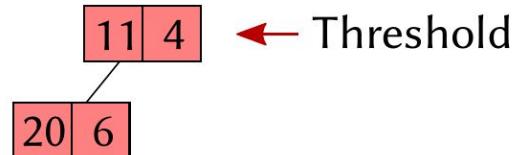


Top- k Retrieval

Term Postings Lists

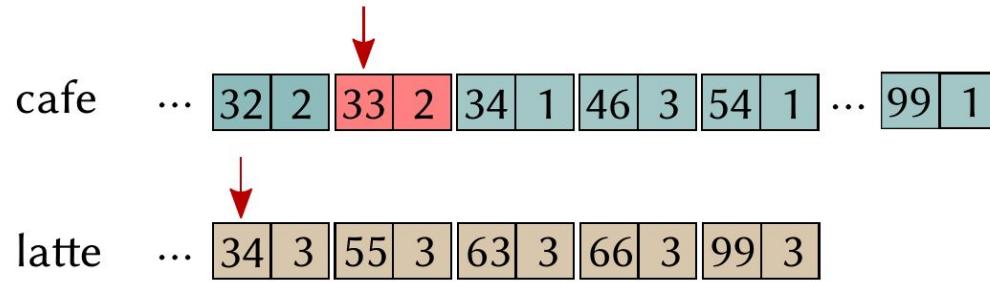


Top- k documents (min-heap) ($k = 2$)

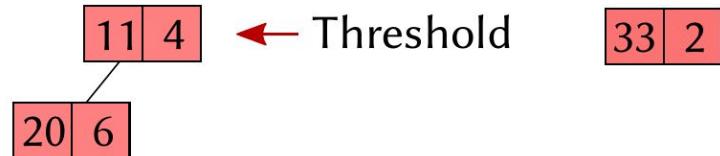


Top- k Retrieval

Term Postings Lists

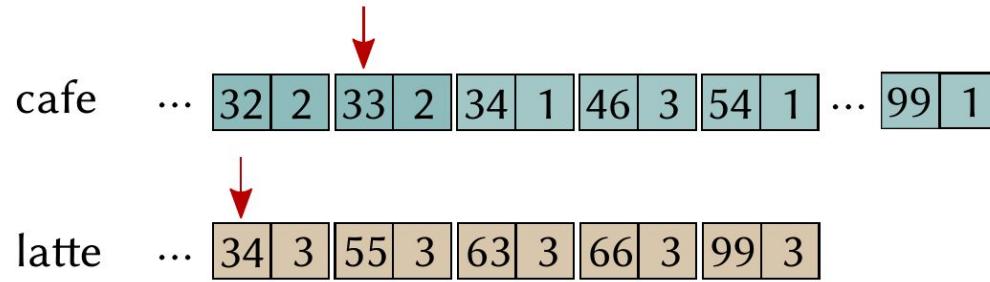


Top- k documents (min-heap) ($k = 2$)

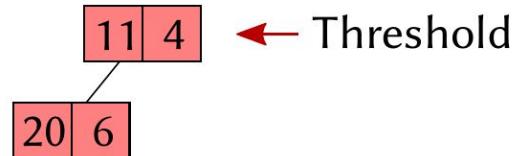


Top- k Retrieval

Term Postings Lists

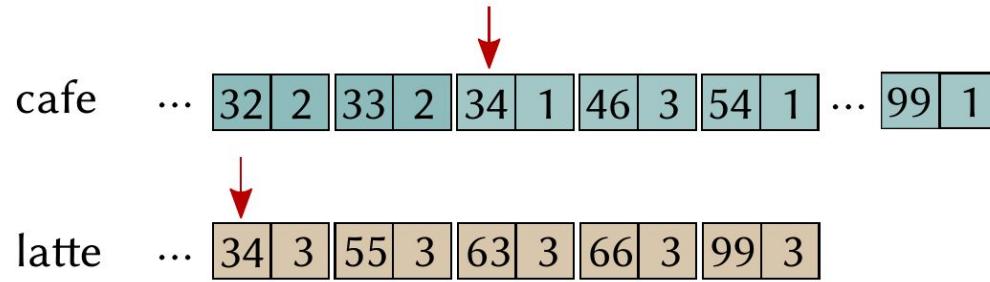


Top- k documents (min-heap) ($k = 2$)

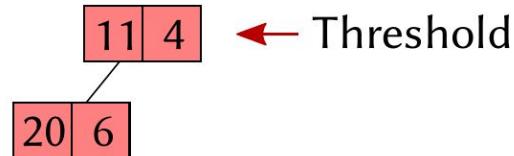


Top- k Retrieval

Term Postings Lists

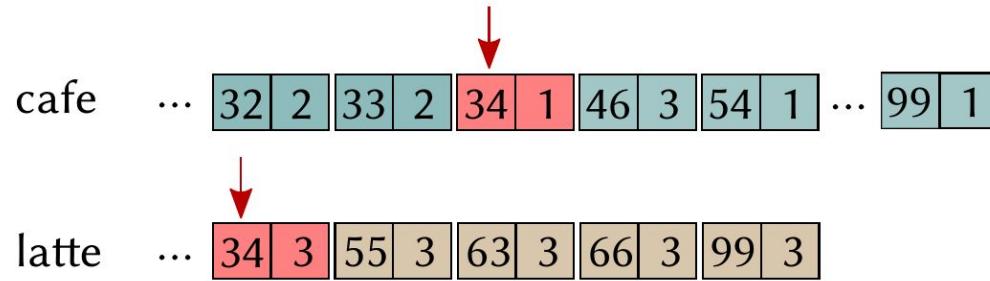


Top- k documents (min-heap) ($k = 2$)

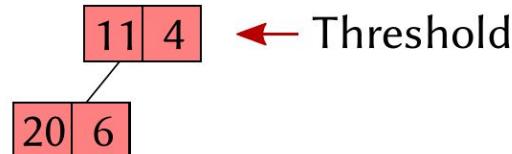


Top- k Retrieval

Term Postings Lists

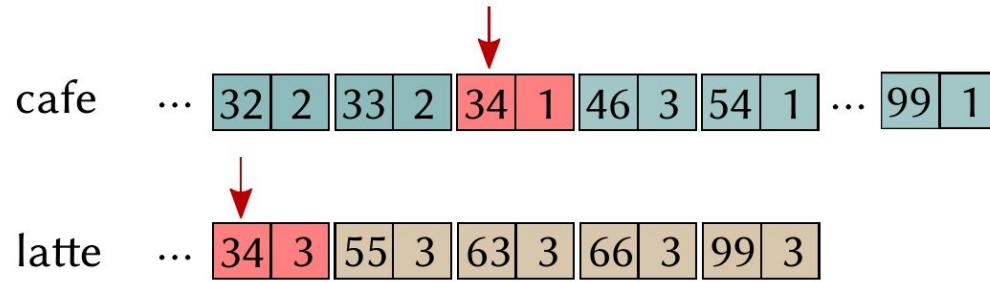


Top- k documents (min-heap) ($k = 2$)

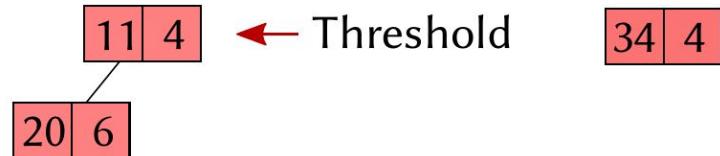


Top- k Retrieval

Term Postings Lists

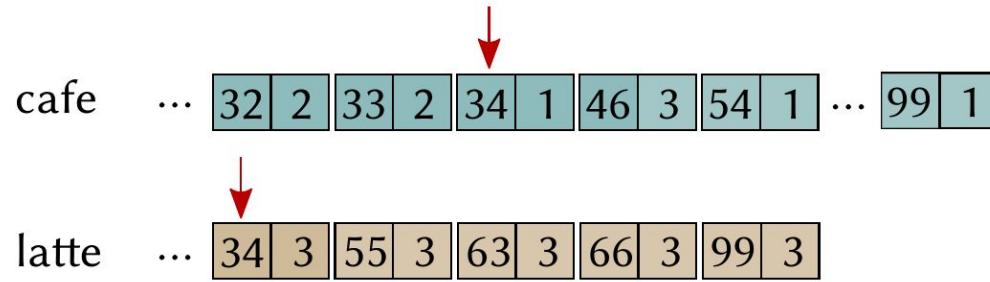


Top- k documents (min-heap) ($k = 2$)

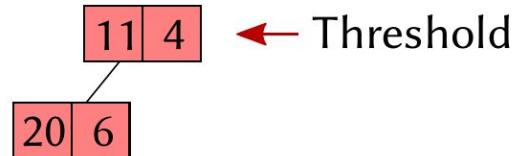


Top- k Retrieval

Term Postings Lists

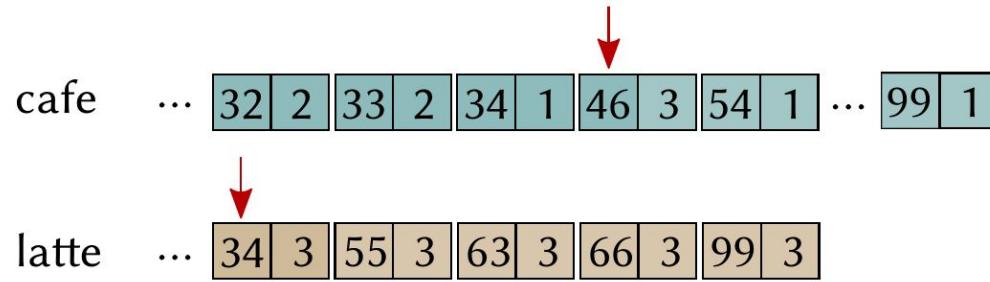


Top- k documents (min-heap) ($k = 2$)

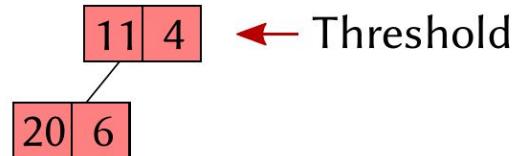


Top- k Retrieval

Term Postings Lists

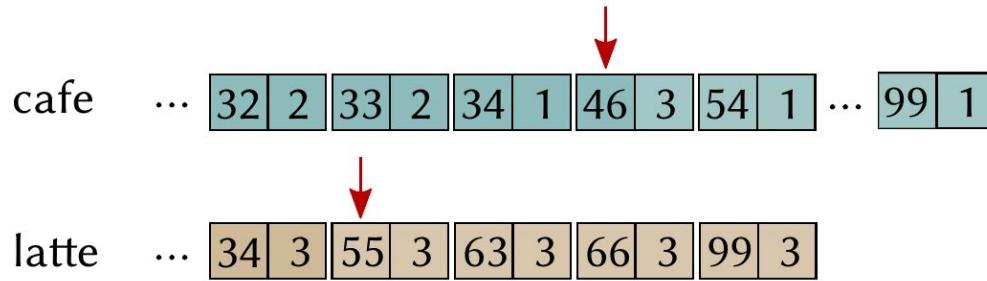


Top- k documents (min-heap) ($k = 2$)

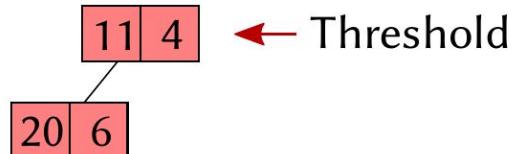


Top- k Retrieval

Term Postings Lists



Top- k documents (min-heap) ($k = 2$)



Top- k Retrieval

- Naive algorithm:
 - Scores *every* single posting!
 - Which means we decompress *every* block of document identifiers **and** impacts.
 - But guaranteed to return the *rank-safe* top- k documents.

Dynamic Pruning

What if we didn't need to score everything in order to get the rank-safe top- k results?

Basic Ingredients

- Our ranking function must be additive;
- We must pre-compute and store the maximum impact that each postings list contains (offline);
- Our index must support efficient random access.

Dynamic Pruning

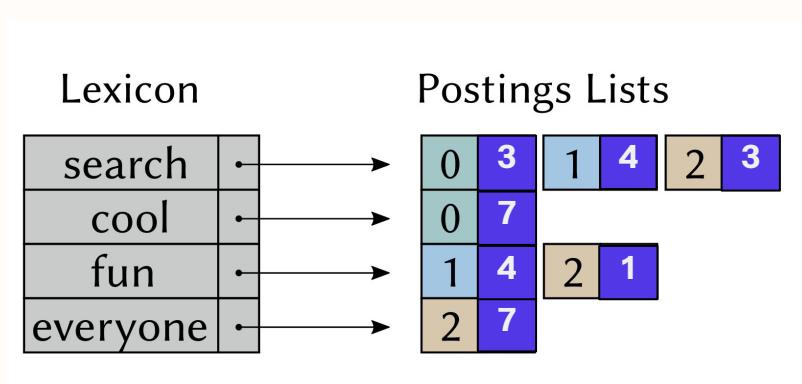
What if we didn't need to score everything in order to get the rank-safe top- k results?

Intuition

- Use the top element of the heap as a *threshold*;
- *Estimate* the score of each document by summing up the list-wise upper-bound scores;
- Only score documents with an estimated score exceeding the heap threshold - bypass otherwise.

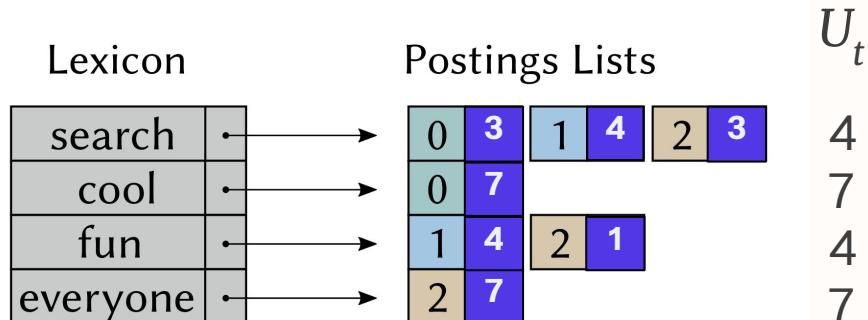
Dynamic Pruning: Index Time

- During indexing, we must pre-compute the list-wise upper-bound score. This is denoted U_t for term t .



Dynamic Pruning: Index Time

- During indexing, we must pre-compute the list-wise upper-bound score. This is denoted U_t for term t .



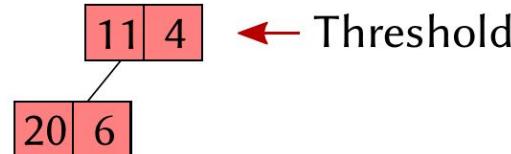
Dynamic Pruning: Query Time

Term Postings Lists

| | | | | | | | | |
|------|-----|--------|--------|--------|--------|--------|-----|--------|
| cafe | ... | 32 2 | 33 2 | 34 1 | 46 3 | 54 1 | ... | 99 1 |
|------|-----|--------|--------|--------|--------|--------|-----|--------|

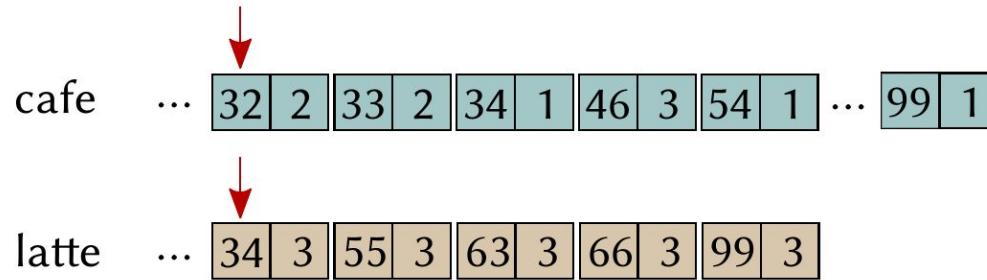
| | | | | | | |
|-------|-----|--------|--------|--------|--------|--------|
| latte | ... | 34 3 | 55 3 | 63 3 | 66 3 | 99 3 |
|-------|-----|--------|--------|--------|--------|--------|

Top- k documents (min-heap) ($k = 2$)

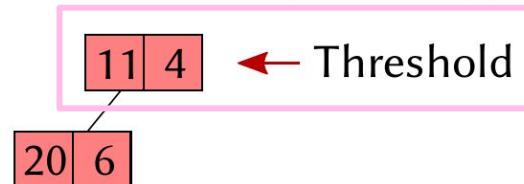


Dynamic Pruning: Query Time

Term Postings Lists



Top- k documents (min-heap) ($k = 2$)

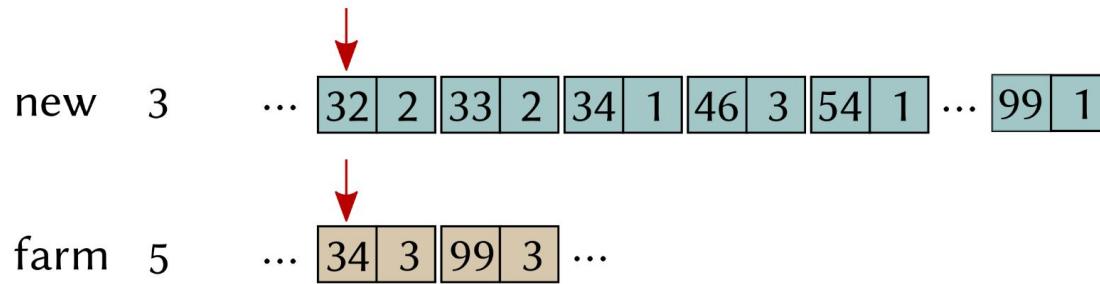


Use the minimum score
in the heap as a
threshold

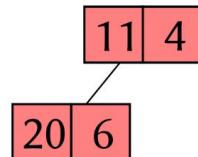
Dynamic Pruning: Walkthrough

Dynamic Pruning: Walkthrough

Term U_t Postings Lists

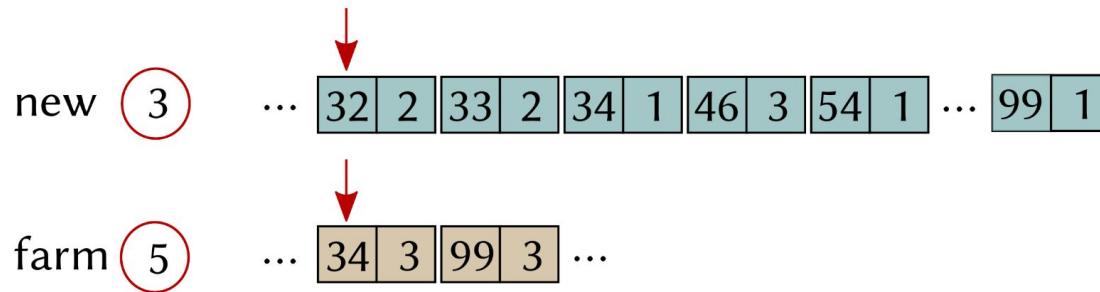


Top- k documents (min-heap) ($k = 2$)

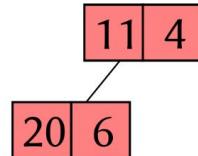


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

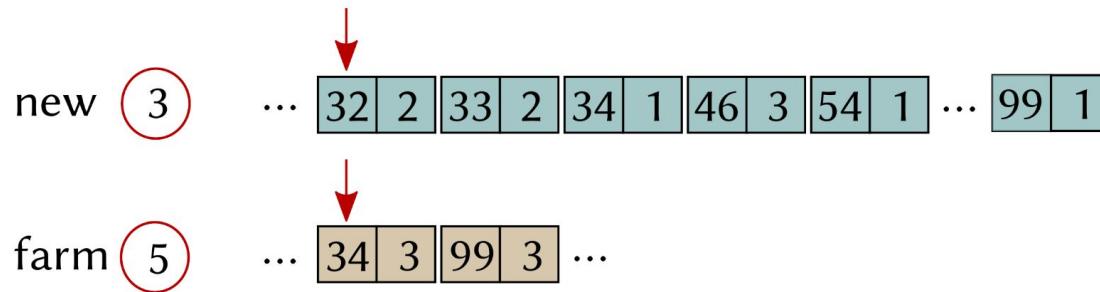


Top- k documents (min-heap) ($k = 2$)

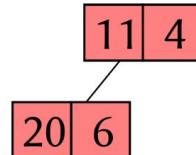


Dynamic Pruning: Walkthrough

Term U_t Postings Lists



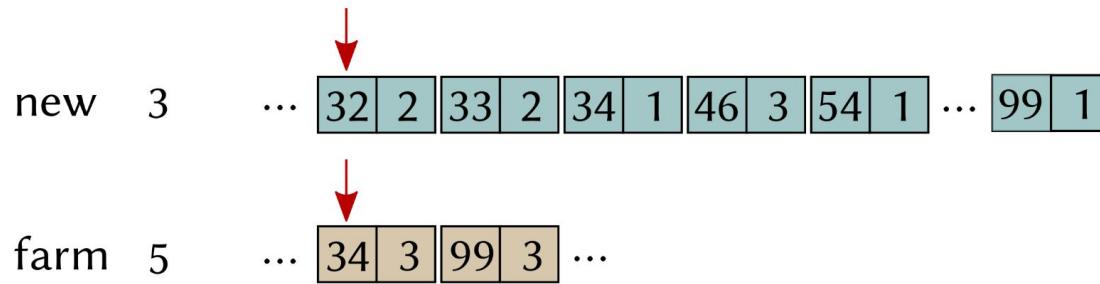
Top- k documents (min-heap) ($k = 2$)



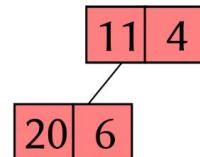
Compute and store at index time

Dynamic Pruning: Walkthrough

Term U_t Postings Lists

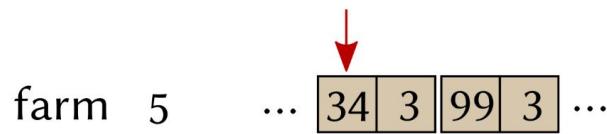
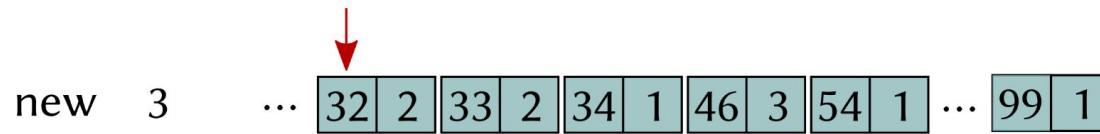


Top- k documents (min-heap) ($k = 2$)

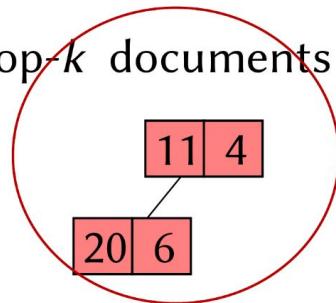


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

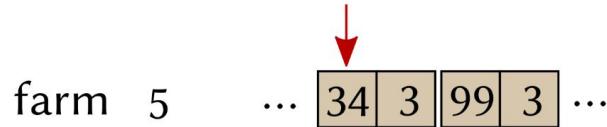
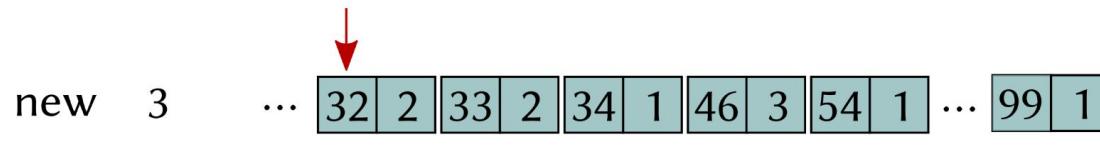


Top- k documents (min-heap) ($k = 2$)

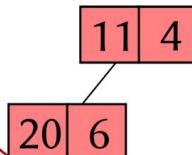


Dynamic Pruning: Walkthrough

Term U_t Postings Lists



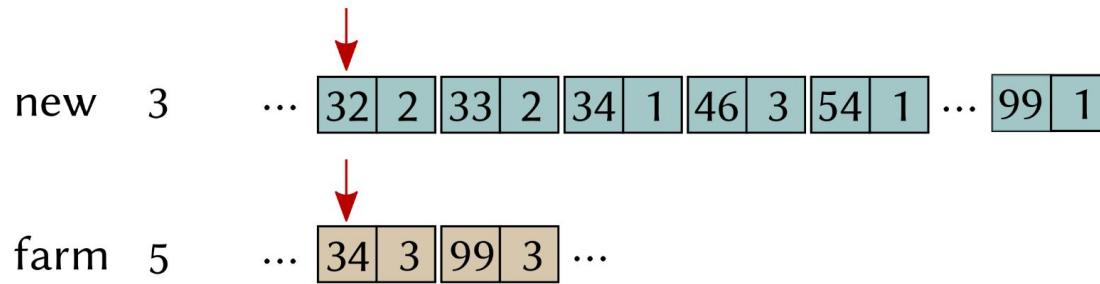
Top- k documents (min-heap) ($k = 2$)



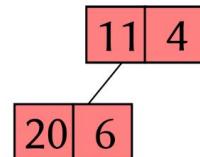
Top 2 documents seen up to this point

Dynamic Pruning: Walkthrough

Term U_t Postings Lists

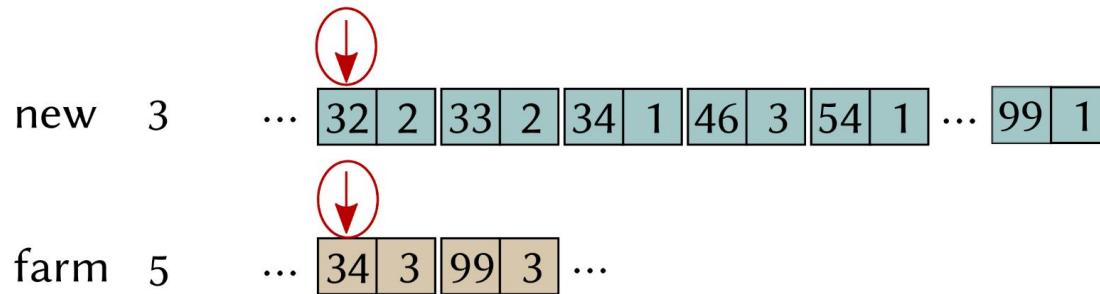


Top- k documents (min-heap) ($k = 2$)

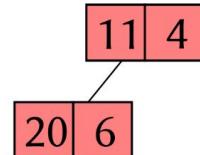


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

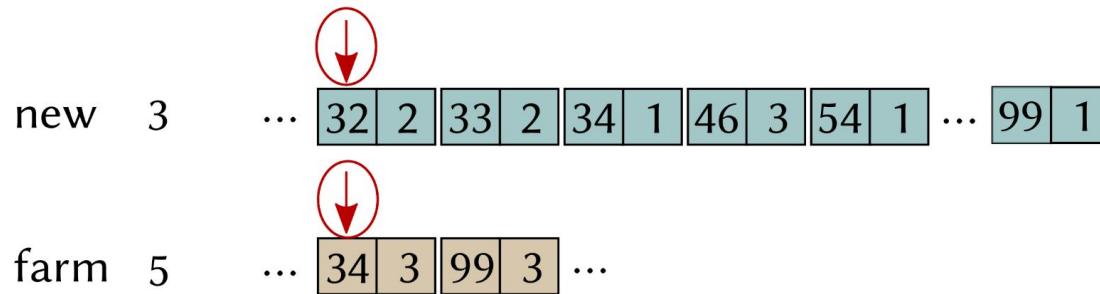


Top- k documents (min-heap) ($k = 2$)

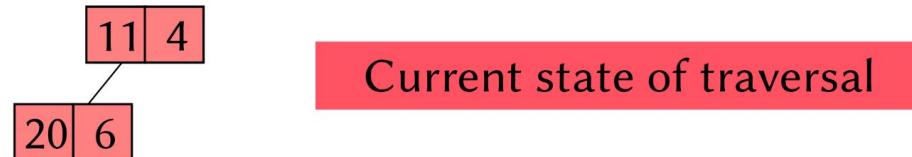


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

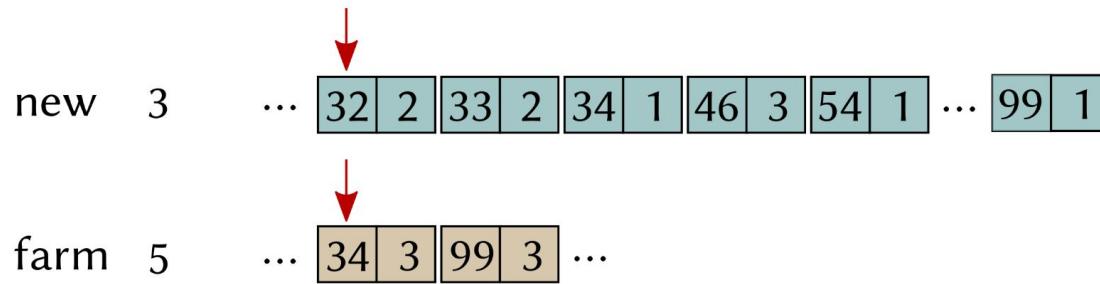


Top- k documents (min-heap) ($k = 2$)

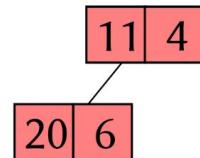


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

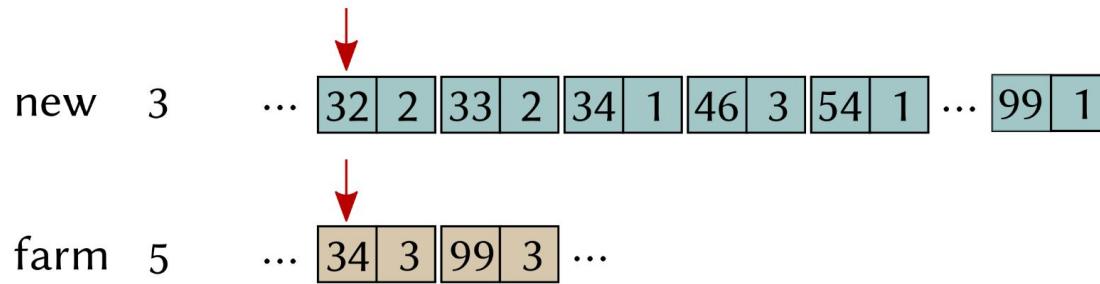


Top- k documents (min-heap) ($k = 2$)

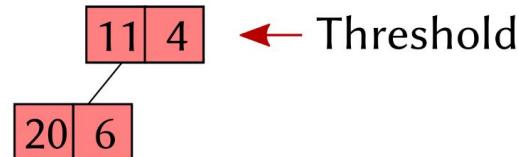


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

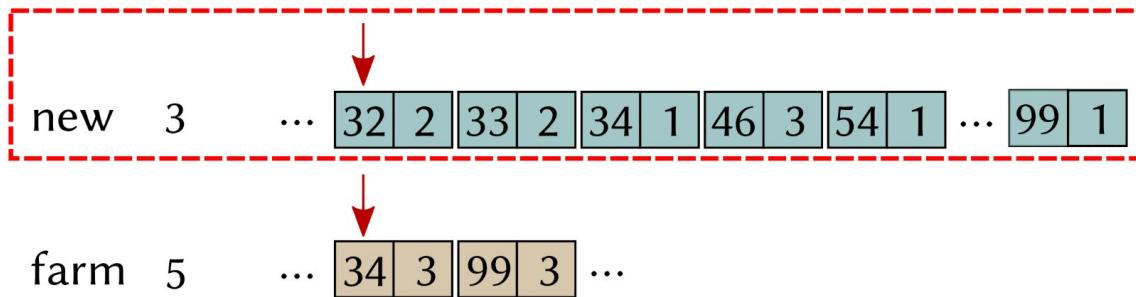


Top- k documents (min-heap) ($k = 2$)

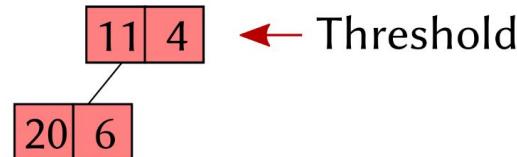


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

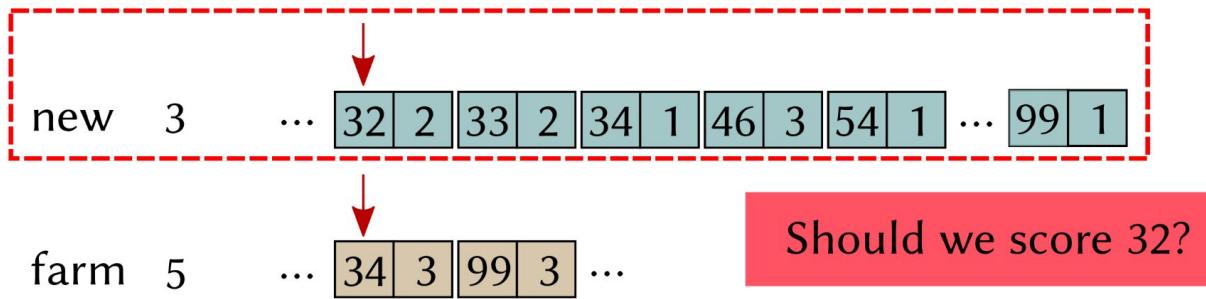


Top- k documents (min-heap) ($k = 2$)

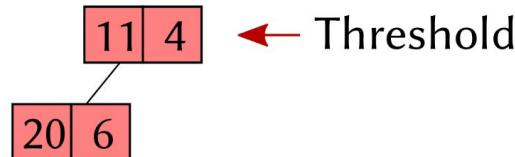


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

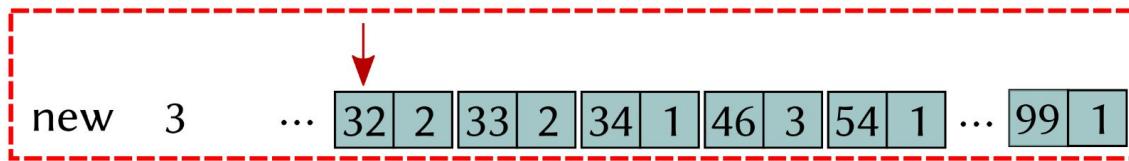


Top- k documents (min-heap) ($k = 2$)



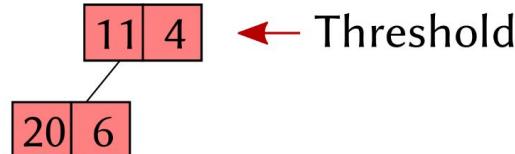
Dynamic Pruning: Walkthrough

Term U_t Postings Lists



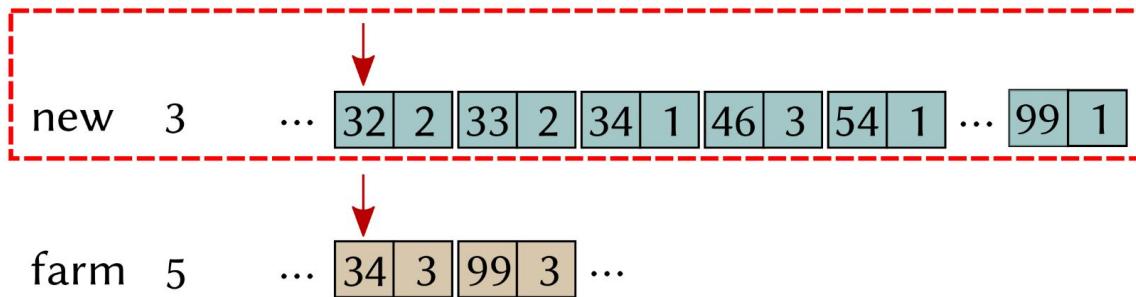
Should we score 32?
3 < Threshold: skip

Top- k documents (min-heap) ($k = 2$)

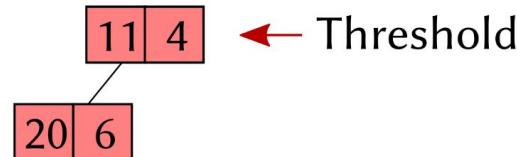


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

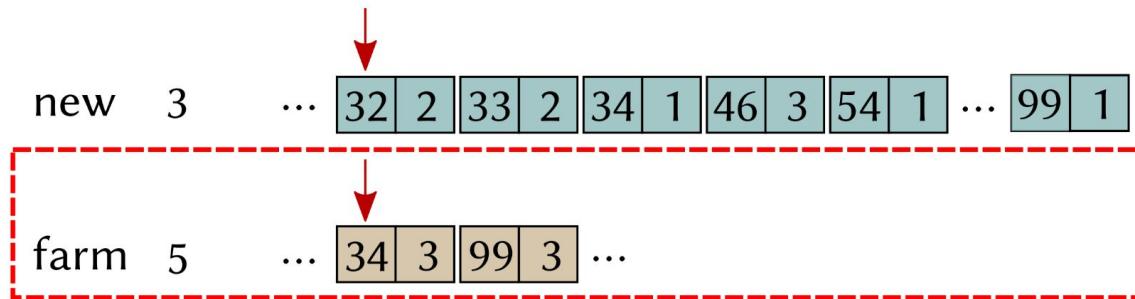


Top- k documents (min-heap) ($k = 2$)

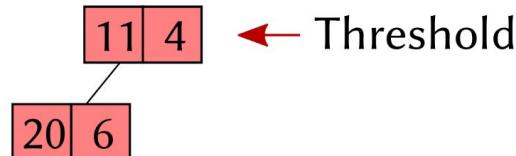


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

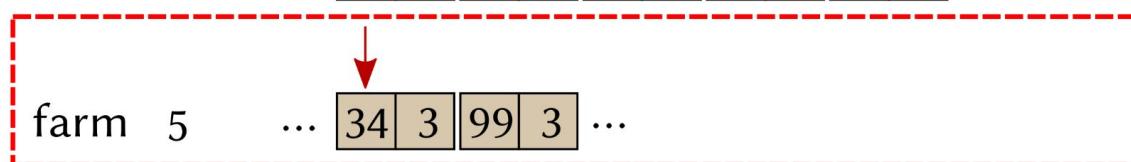
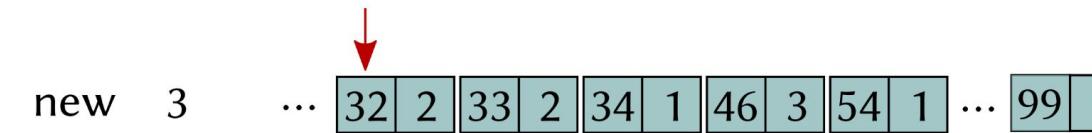


Top- k documents (min-heap) ($k = 2$)

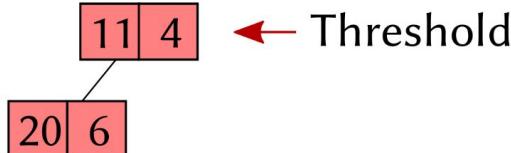


Dynamic Pruning: Walkthrough

Term U_t Postings Lists



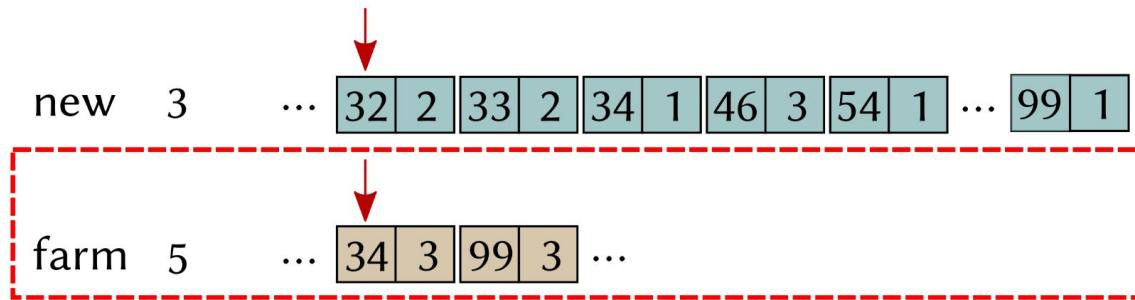
Top- k documents (min-heap) ($k = 2$)



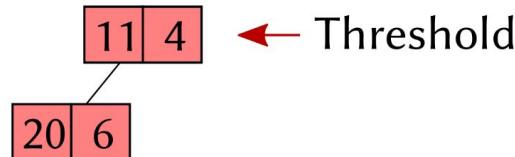
Should we score 34?
 $3 + 5 >$ Threshold: Yes

Dynamic Pruning: Walkthrough

Term U_t Postings Lists

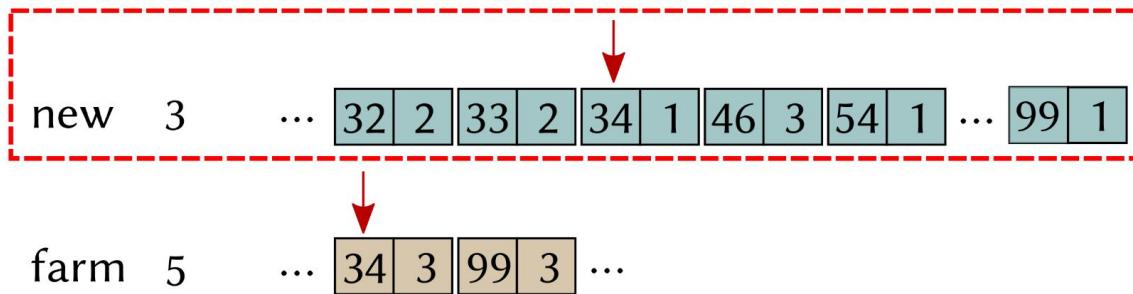


Top- k documents (min-heap) ($k = 2$)

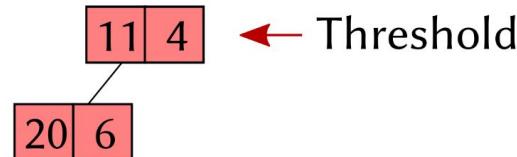


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

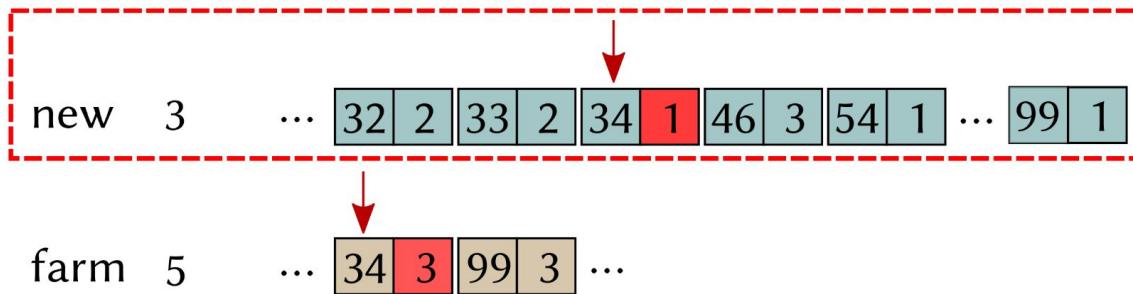


Top- k documents (min-heap) ($k = 2$)

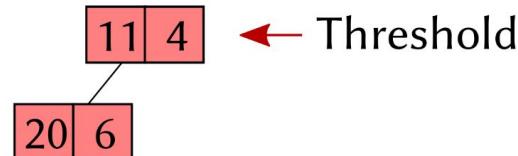


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

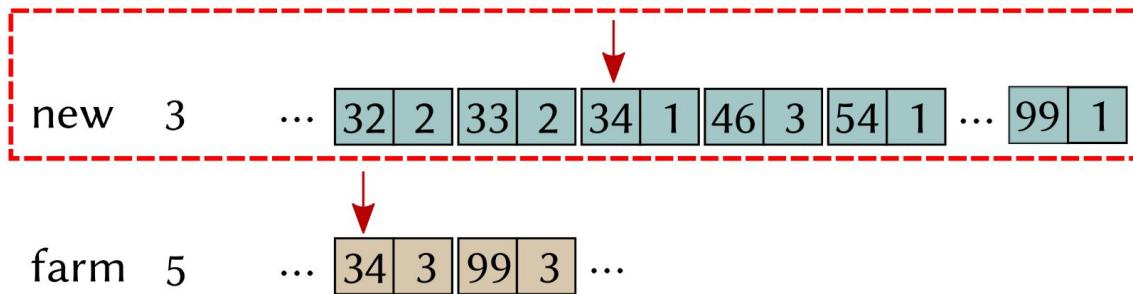


Top- k documents (min-heap) ($k = 2$)

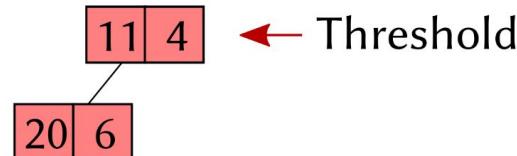


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

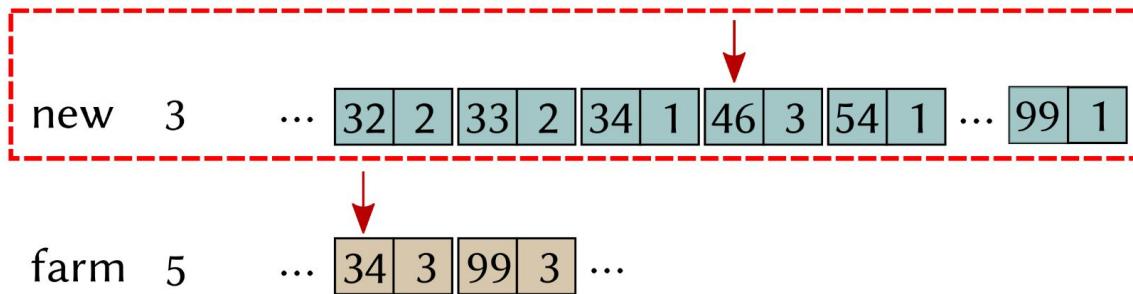


Top- k documents (min-heap) ($k = 2$)

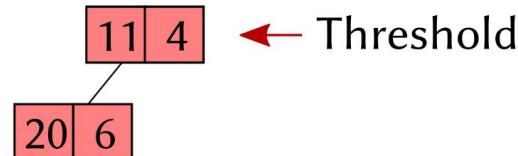


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

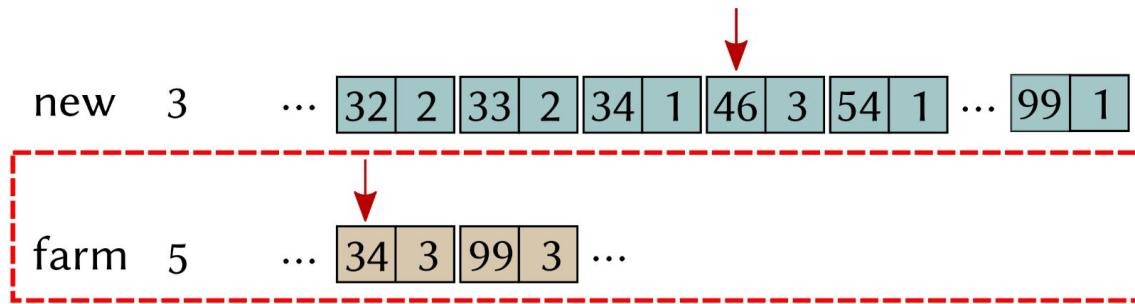


Top- k documents (min-heap) ($k = 2$)

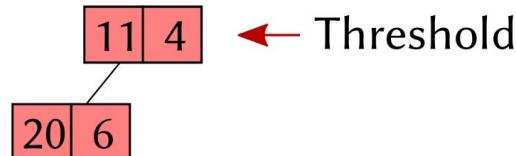


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

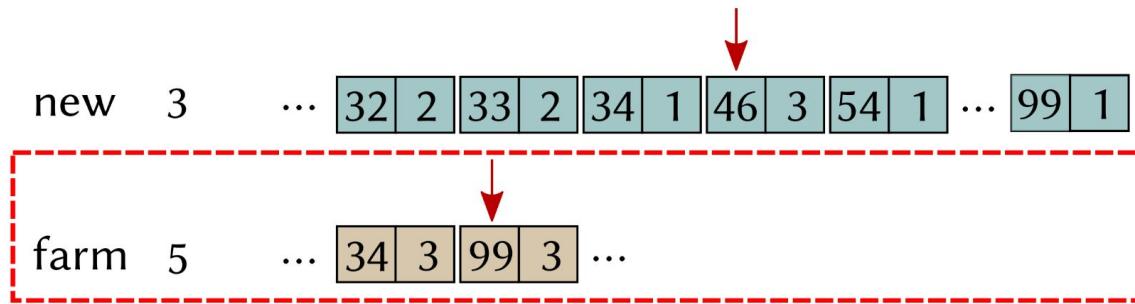


Top- k documents (min-heap) ($k = 2$)

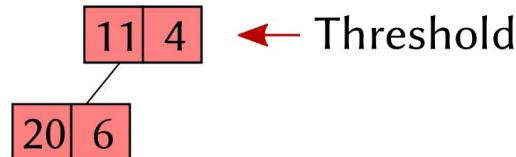


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

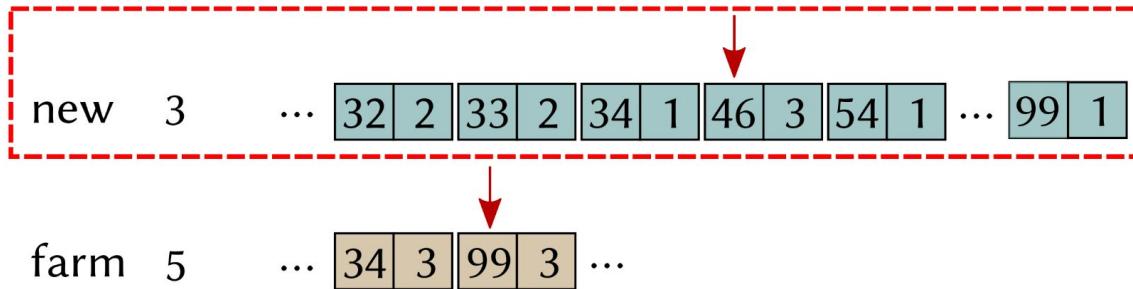


Top- k documents (min-heap) ($k = 2$)

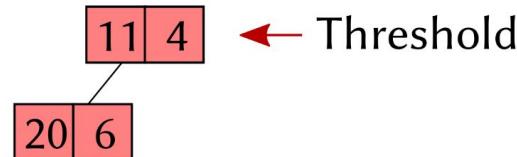


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

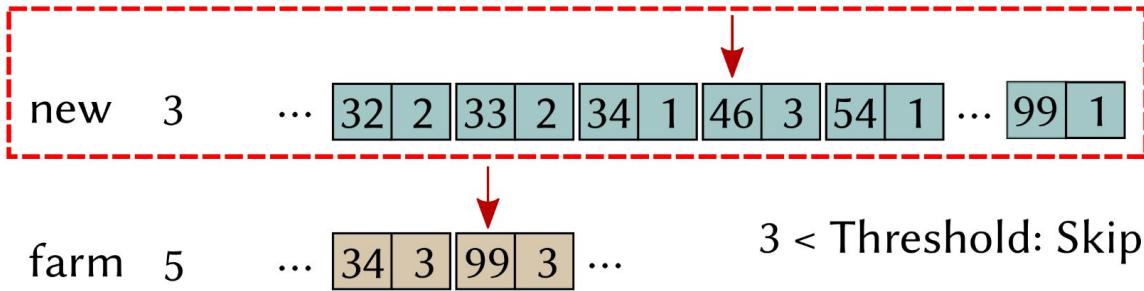


Top- k documents (min-heap) ($k = 2$)

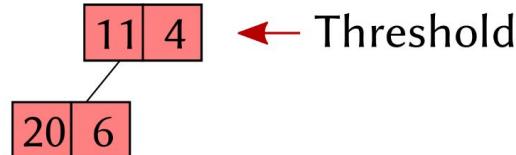


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

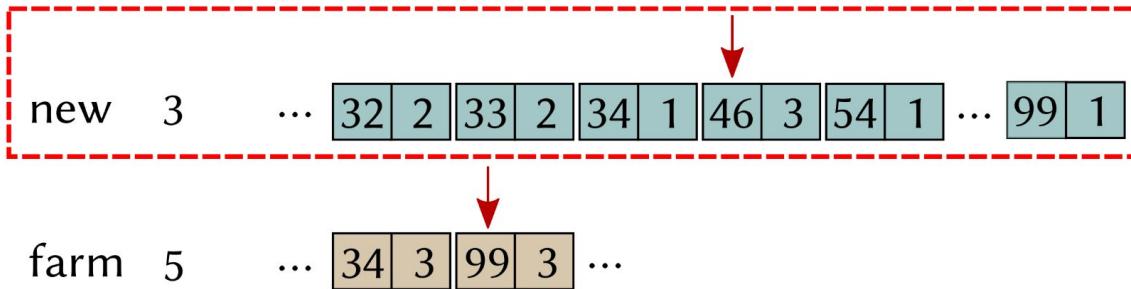


Top- k documents (min-heap) ($k = 2$)

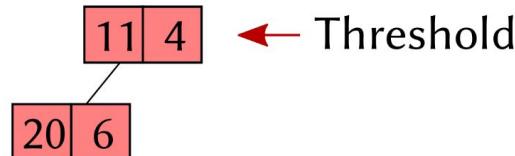


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

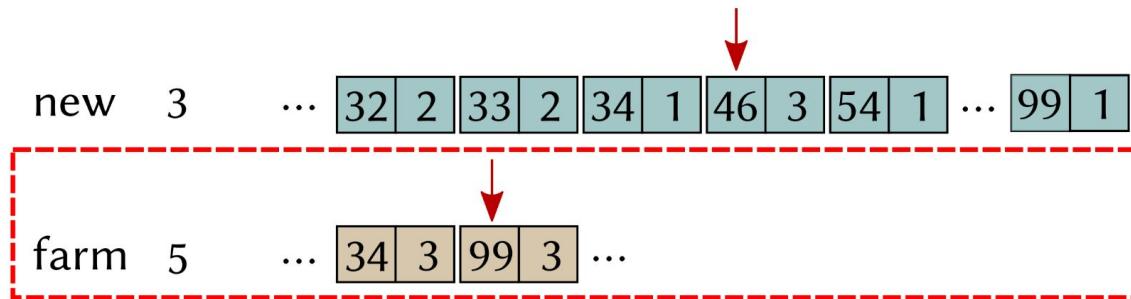


Top- k documents (min-heap) ($k = 2$)

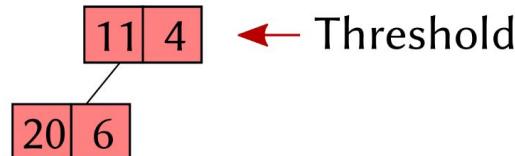


Dynamic Pruning: Walkthrough

Term U_t Postings Lists

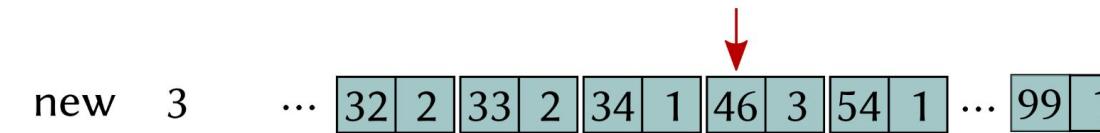


Top- k documents (min-heap) ($k = 2$)



Dynamic Pruning: Walkthrough

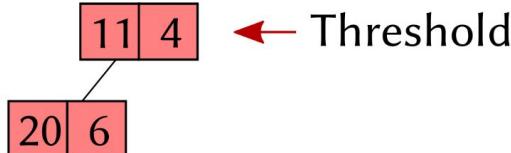
Term U_t Postings Lists



farm 5 ... [34 3] [99 3] ...

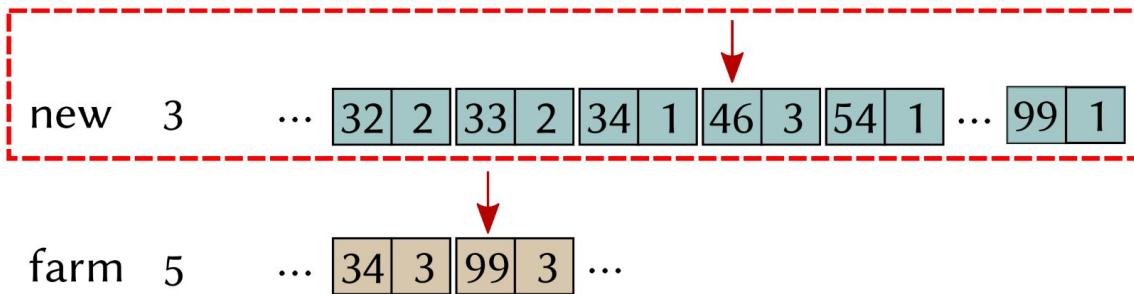
Top- k documents (min-heap) ($k = 2$)

$3 + 5 > \text{Threshold}$
Score 99



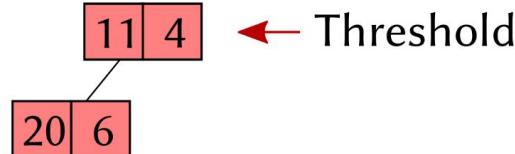
Dynamic Pruning: Walkthrough

Term U_t Postings Lists



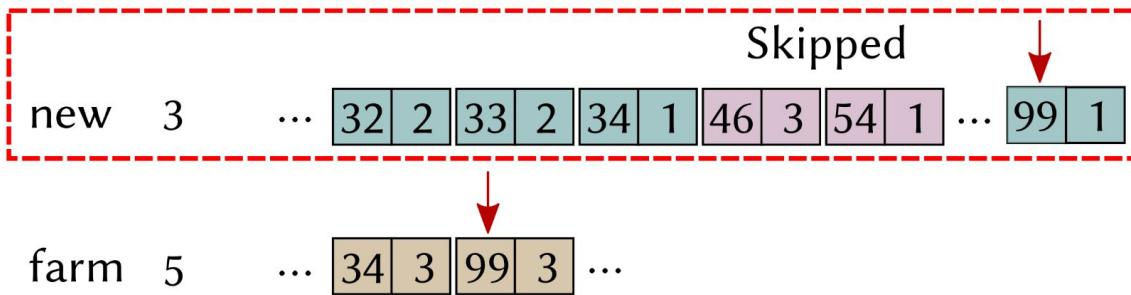
Top- k documents (min-heap) ($k = 2$)

$3 + 5 > \text{Threshold}$
Score 99



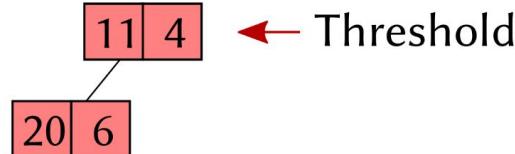
Dynamic Pruning: Walkthrough

Term U_t Postings Lists



Top- k documents (min-heap) ($k = 2$)

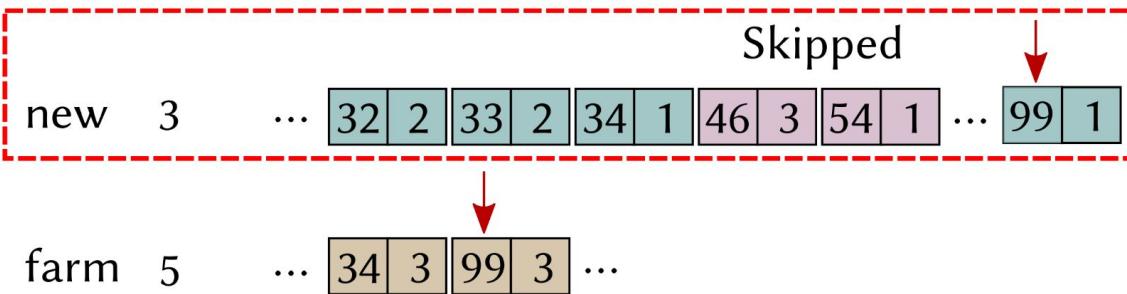
$3 + 5 > \text{Threshold}$
Score 99



Dynamic Pruning: Walkthrough

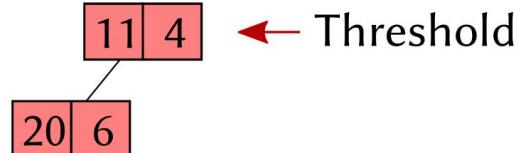
Term U_t Postings Lists

How does the skipping work?



Top- k documents (min-heap) ($k = 2$)

$3 + 5 > \text{Threshold}$
Score 99



Next-GEQ Operator

Dynamic pruning algorithms depend on the **efficient** implementation of the *next_geq()* operator.

next_geq(n) forwards the postings list cursor to document *n*, if it exists, or the next greater document.

In our block-based index, we retain an uncompressed document identifier such that we can skip to the candidate block efficiently; then we must seek within the block.

Next-GEQ Operator

pisa/include/pisa/block_posting_list.hpp - Line 127

```
/**  
 * Moves to the next document, counting from the current position,  
 * with the ID equal to or greater than `lower_bound`.  
 *  
 * In particular, if called with a value that is less than or equal  
 * to the current document ID, the position will not change.  
 */  
void PISA_ALWAYSINLINE next_geq(uint64_t lower_bound) {  
    if PISA_UNLIKELY (lower_bound > m_cur_block_max) {  
        // binary search seems to perform worse here  
        if (lower_bound > block_max(m_blocks - 1)) {  
            m_cur_docid = m_universe;  
            return;  
        }  
  
        uint64_t block = m_cur_block + 1;  
        while (block_max(block) < lower_bound) {  
            ++block;  
        }  
  
        decode_docs_block(block);  
    }  
  
    while (docid() < lower_bound) {  
        m_cur_docid += m_docs_buf[+m_pos_in_block] + 1;  
        assert(m_pos_in_block < m_cur_block_size);  
    }  
}
```

Next-GEQ Operator

```
/**  
 * Moves to the next document, counting from the current position,  
 * with the ID equal to or greater than `lower_bound`.  
 *  
 * In particular, if called with a value that is less than or equal  
 * to the current document ID, the position will not change.  
 */  
void PISA_ALWAYSINLINE next_geq(uint64_t lower_bound) {  
    if PISA_UNLIKELY (lower_bound > m_cur_block_max) {  
        // binary search seems to perform worse here  
        if (lower_bound > block_max(m_blocks - 1)) {  
            m_cur_docid = m_universe;  
            return;  
        }  
  
        uint64_t block = m_cur_block + 1;  
        while (block_max(block) < lower_bound) {  
            ++block;  
        }  
  
        decode_docs_block(block);  
    }  
  
    while (docid() < lower_bound) {  
        m_cur_docid += m_docs_buf[+m_pos_in_block] + 1;  
        assert(m_pos_in_block < m_cur_block_size);  
    }  
}
```

If the element we are searching for is in a different block than the current one...

Next-GEQ Operator

```
/**  
 * Moves to the next document, counting from the current position,  
 * with the ID equal to or greater than `lower_bound`.  
 *  
 * In particular, if called with a value that is less than or equal  
 * to the current document ID, the position will not change.  
 */  
void PISA_ALWAYSINLINE next_geq(uint64_t lower_bound) {  
    if PISA_UNLIKELY (lower_bound > m_cur_block_max) {  
        // binary search seems to perform worse here  
        if (lower_bound > block_max(m_blocks - 1)) {  
            m_cur_docid = m_universe;  
            return;  
        }  
  
        uint64_t block = m_cur_block + 1;  
        while (block_max(block) < lower_bound) {  
            ++block;  
        }  
  
        decode_docs_block(block);  
    }  
  
    while (docid() < lower_bound) {  
        m_cur_docid += m_docs_buf[++m_pos_in_block] + 1;  
        assert(m_pos_in_block < m_cur_block_size);  
    }  
}
```

This means the element we are searching for is larger than the largest element in this postings list. So, we return.

Next-GEQ Operator

```
/**  
 * Moves to the next document, counting from the current position,  
 * with the ID equal to or greater than `lower_bound`.  
 *  
 * In particular, if called with a value that is less than or equal  
 * to the current document ID, the position will not change.  
 */  
void PISA_ALWAYSINLINE next_geq(uint64_t lower_bound) {  
    if PISA_UNLIKELY (lower_bound > m_cur_block_max) {  
        // binary search seems to perform worse here  
        if (lower_bound > block_max(m_blocks - 1)) {  
            m_cur_docid = m_universe;  
            return;  
        }  
  
        uint64_t block = m_cur_block + 1;  
        while (block_max(block) < lower_bound) {  
            ++block;  
        }  
  
        decode_docs_block(block);  
    }  
  
    while (docid() < lower_bound) {  
        m_cur_docid += m_docs_buf[++m_pos_in_block] + 1;  
        assert(m_pos_in_block < m_cur_block_size);  
    }  
}
```

Walk across the uncompressed structure that stores the maximum identifier in each block until we exceed the target element

Next-GEQ Operator

```
/**  
 * Moves to the next document, counting from the current position,  
 * with the ID equal to or greater than `lower_bound`.  
 *  
 * In particular, if called with a value that is less than or equal  
 * to the current document ID, the position will not change.  
 */  
void PISA_ALWAYSINLINE next_geq(uint64_t lower_bound) {  
    if PISA_UNLIKELY (lower_bound > m_cur_block_max) {  
        // binary search seems to perform worse here  
        if (lower_bound > block_max(m_blocks - 1)) {  
            m_cur_docid = m_universe;  
            return;  
        }  
  
        uint64_t block = m_cur_block + 1;  
        while (block_max(block) < lower_bound) {  
            ++block;  
        }  
        decode_docs_block(block);  
    }  
  
    while (docid() < lower_bound) {  
        m_cur_docid += m_docs_buf[++m_pos_in_block] + 1;  
        assert(m_pos_in_block < m_cur_block_size);  
    }  
}
```

Decode the current block into a buffer - this block must contain the target element if it exists...

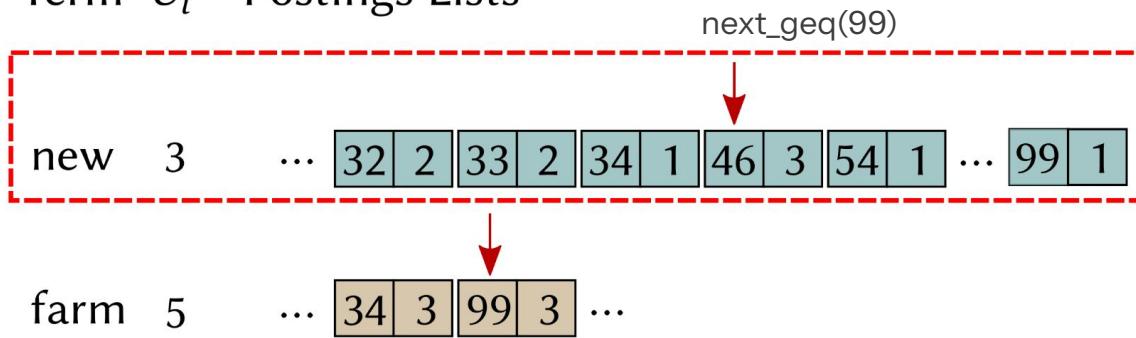
Next-GEQ Operator

```
/**  
 * Moves to the next document, counting from the current position,  
 * with the ID equal to or greater than `lower_bound`.  
 *  
 * In particular, if called with a value that is less than or equal  
 * to the current document ID, the position will not change.  
 */  
void PISA_ALWAYSINLINE next_geq(uint64_t lower_bound) {  
    if PISA_UNLIKELY (lower_bound > m_cur_block_max) {  
        // binary search seems to perform worse here  
        if (lower_bound > block_max(m_blocks - 1)) {  
            m_cur_docid = m_universe;  
            return;  
        }  
  
        uint64_t block = m_cur_block + 1;  
        while (block_max(block) < lower_bound) {  
            ++block;  
        }  
  
        decode_docs_block(block);  
    }  
  
    while (docid() < lower_bound) {  
        m_cur_docid += m_docs_buf[++m_pos_in_block] + 1;  
        assert(m_pos_in_block < m_cur_block_size);  
    }  
}
```

Now search within the block to find the element (or the next greater one)

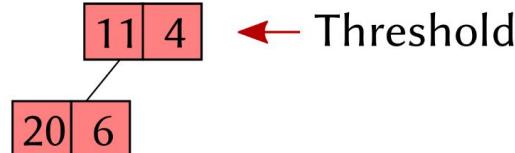
Next-GEQ Operator

Term U_t Postings Lists



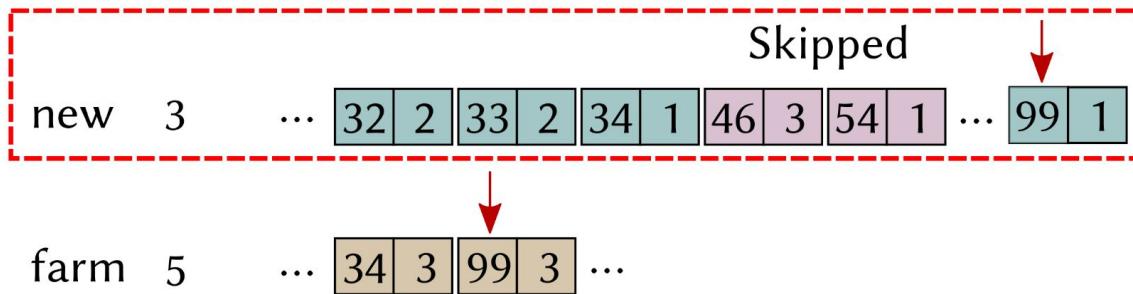
Top- k documents (min-heap) ($k = 2$)

$3 + 5 > \text{Threshold}$
Score 99



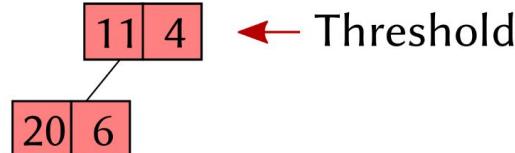
Next-GEQ Operator

Term U_t Postings Lists



Top- k documents (min-heap) ($k = 2$) $3 + 5 > \text{Threshold}$

Score 99



Skipping - Saving Work

We do not need to decompress any blocks we skip!

We do not need to score any documents within any of those blocks either!

But we *do* pay **overhead** deciding whether to skip or not.

We also need to be **very careful** with which cursor moves ahead first. More on this shortly...

Dynamic Pruning Algorithms

Efficient Query Evaluation using a Two-Level Retrieval Process

Andrei Z. Broder[§], David Carmel*, Michael Herscovici*, Aya Soffer*, Jason Zien[†]

([§])IBM Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532

(*)IBM Research Lab in Haifa, MATAM, Haifa 31905, ISRAEL

([†])IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120

WAND - “Weak” or “Weighted” AND
CIKM 2003!

Dynamic Pruning Algorithms

QUERY EVALUATION: STRATEGIES AND OPTIMIZATIONS

HOWARD TURTLE and JAMES FLOOD

West Publishing Co., 610 Opperman Drive, Eagan, MN 55123, U.S.A.

(Received January 1995; accepted in final form March 1995)

Abstract—This paper discusses the two major query evaluation strategies used in large text retrieval systems and analyzes the performance of these strategies. We then discuss several optimization techniques that can be used to reduce evaluation costs and present simulation results to compare the performance of these optimization techniques when evaluating natural language queries with a collection of full text legal materials.

MaxScore

IPM 1995!

Dynamic Pruning Algorithms

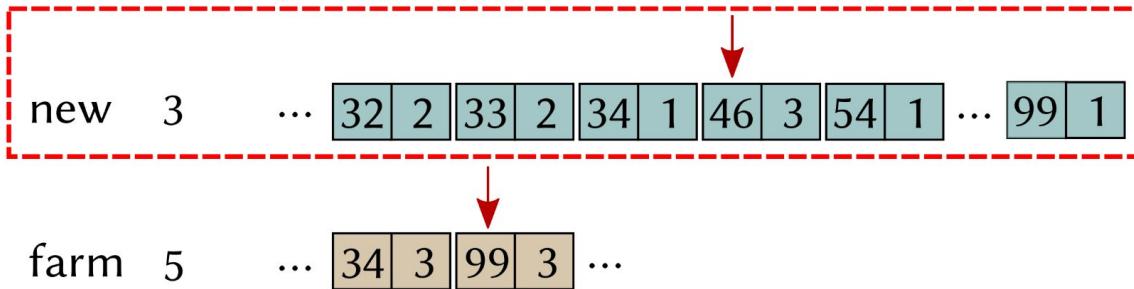
WAND and MaxScore are the two main families of document-at-a-time dynamic pruning algorithms.

Our worked example was inspired by the WAND algorithm.

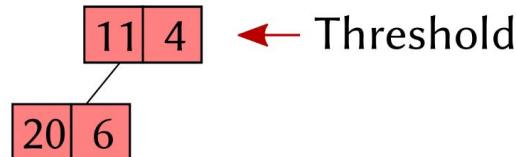
With WAND, we must ensure the cursors are always processed in *ascending order of their current identifier*.

Dynamic Pruning: Walkthrough

Term U_t Postings Lists

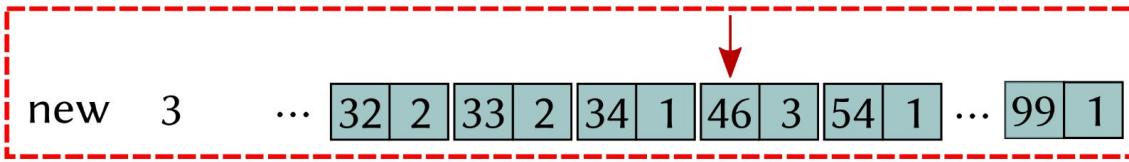


Top- k documents (min-heap) ($k = 2$)



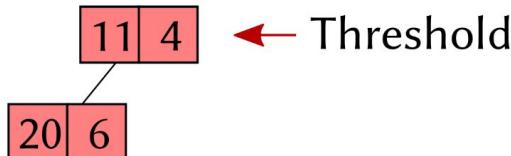
Dynamic Pruning: Walkthrough

Term U_t Postings Lists



farm 5 ... [34, 3] [99, 3] ...

Top- k documents (min-heap) ($k = 2$)



Invariant: The current identifier under each cursor must monotonically increase as we move down through the cursors.

That is, “the element being pointed to in list at the top must be less than or equal to the element being pointed to in the list at the bottom”

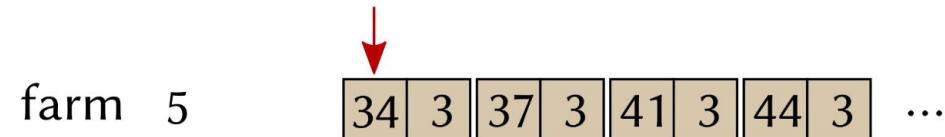
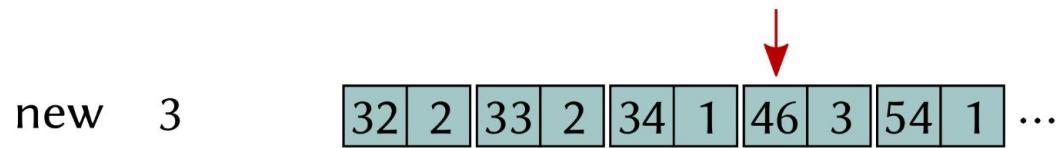
Dynamic Pruning Algorithms

With WAND, we must ensure the cursors are always processed in *ascending order of their current identifier.*

This means we may need to **sort** the cursors during processing!

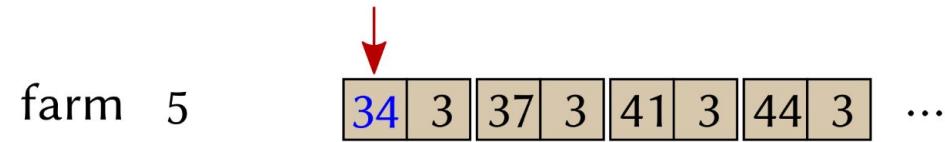
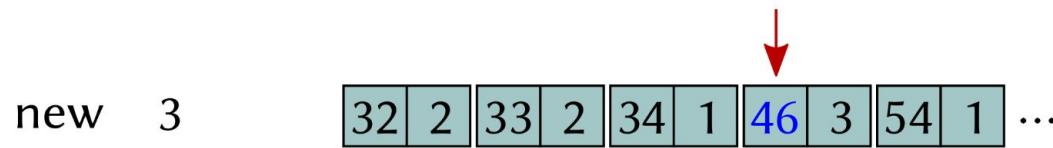
WAND: Sorting Cursors

Term U_t Postings Lists



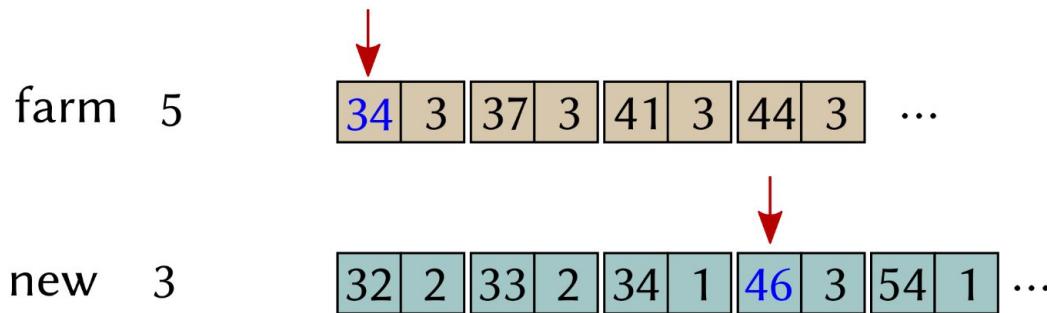
WAND: Sorting Cursors

Term U_t Postings Lists

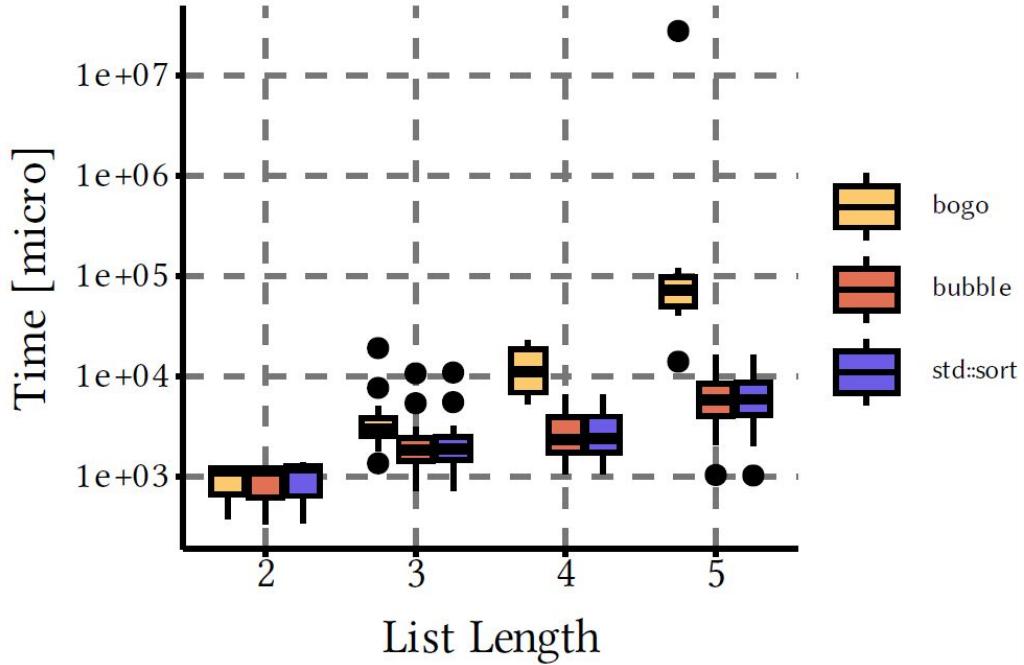


WAND: Sorting Cursors

Term U_t Postings Lists



WAND: Sorting Cursors



Just for fun... What happens if we change the sorting algorithm?

MaxScore: No Sorting Required!

Instead of sorting the cursors before each iteration,
MaxScore sorts the lists **once** before processing begins.

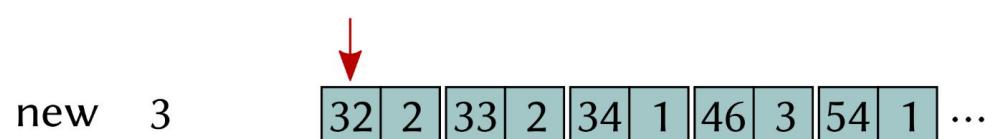
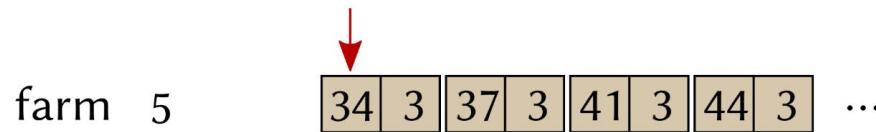
Typically, this sort is ascending on the list upper bounds.

Can also sort on the posting list lengths - either works.

MaxScore: No Sorting Required!

Instead of sorting the cursors before each iteration,
MaxScore sorts the lists **once** before processing begins.

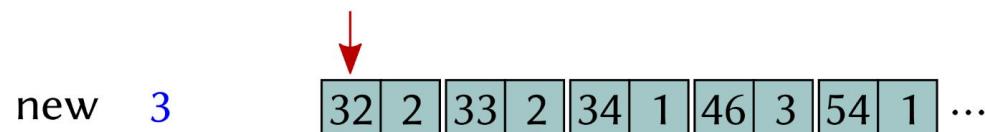
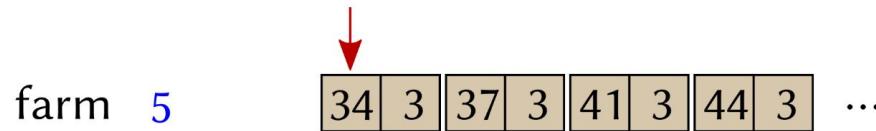
Term U_t Postings Lists



MaxScore: No Sorting Required!

Instead of sorting the cursors before each iteration,
MaxScore sorts the lists **once** before processing begins.

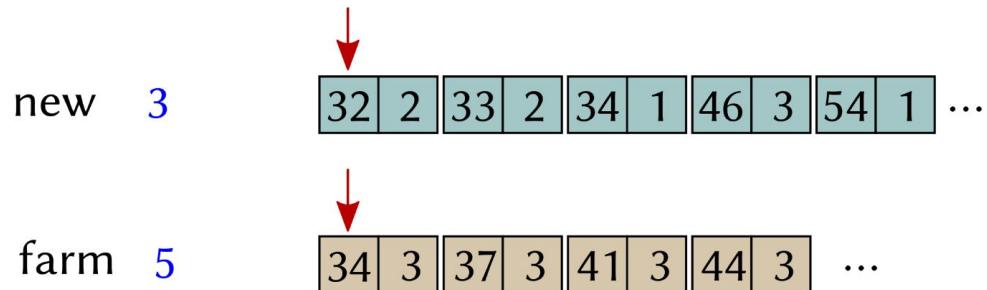
Term U_t Postings Lists



MaxScore: No Sorting Required!

Instead of sorting the cursors before each iteration,
MaxScore sorts the lists **once** before processing begins.

Term U_t Postings Lists



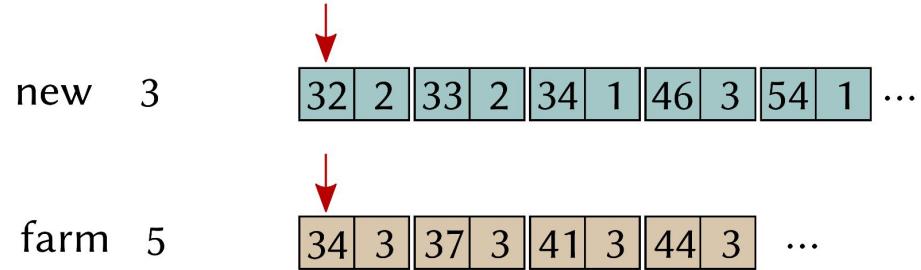
MaxScore: No Sorting Required!

Instead of sorting the cursors before each iteration,
MaxScore sorts the lists **once** before processing begins.

We then compute the *prefix sum* of the upper-bound scores.

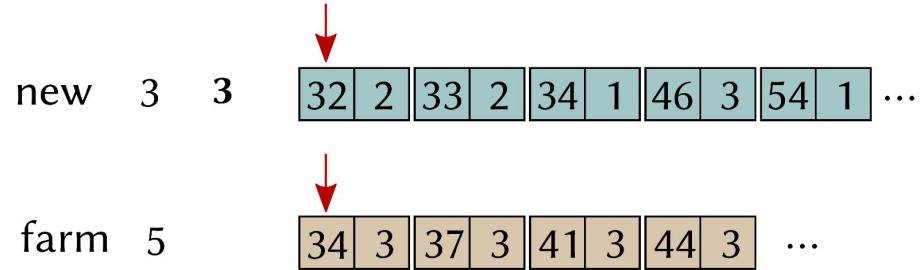
MaxScore: No Sorting Required!

Term U_t PS Postings List



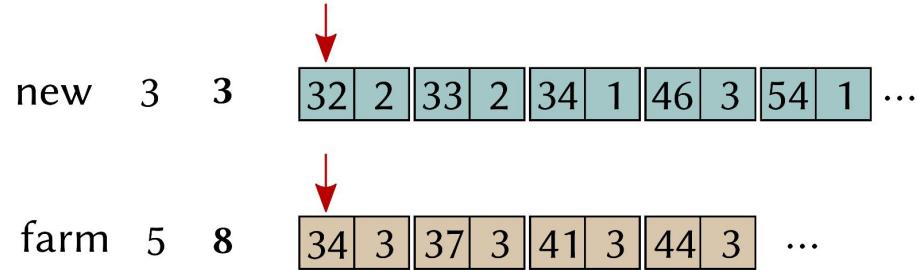
MaxScore: No Sorting Required!

Term U_t PS Postings List



MaxScore: No Sorting Required!

Term U_t PS Postings List



MaxScore: No Sorting Required!

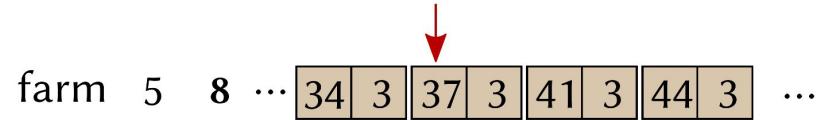
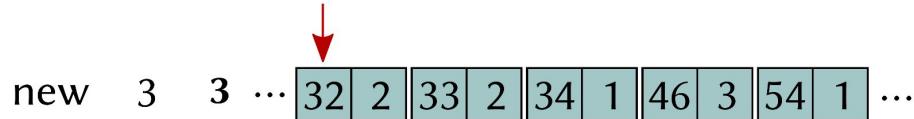
During processing, we split the cursors into two logical sets:

- **Essential Lists:** A document must have at least one essential term to be considered for scoring.
- **Non-Essential Lists:** Are ignored until a final document score needs to be computed.

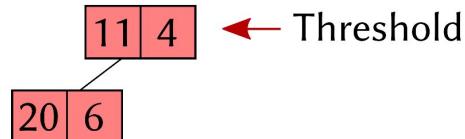
We only iterate over the *essential* lists, allowing us to skip documents that have no hope of entering the results list.

MaxScore: No Sorting Required!

Term U_t PS Postings List

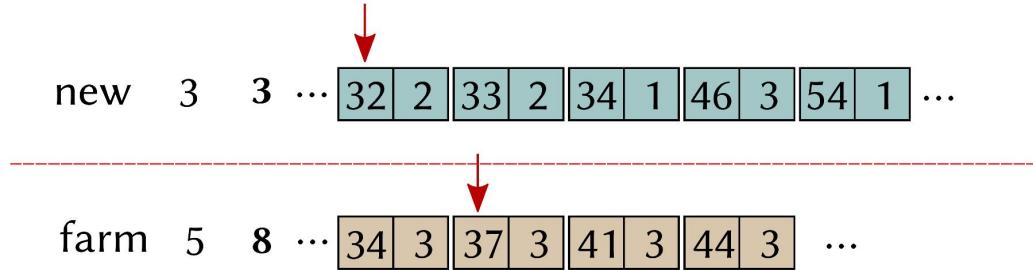


Top- k documents (min-heap) ($k = 2$)

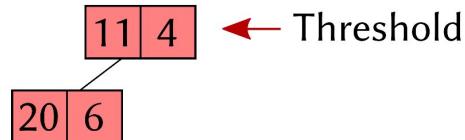


MaxScore: No Sorting Required!

Term U_t PS Postings List

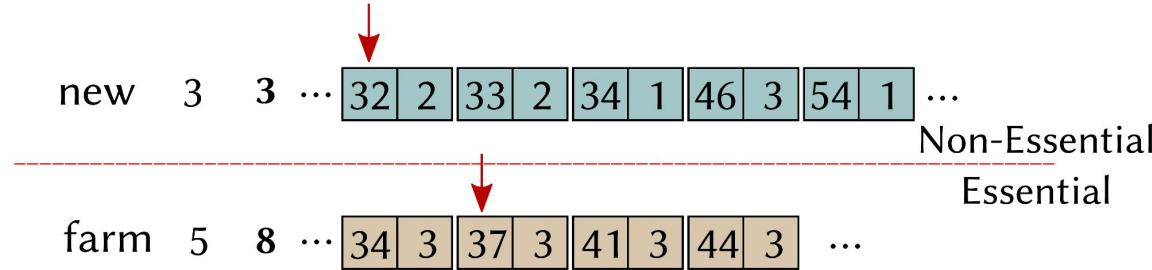


Top- k documents (min-heap) ($k = 2$)

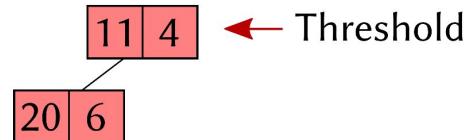


MaxScore: No Sorting Required!

Term U_t PS Postings List

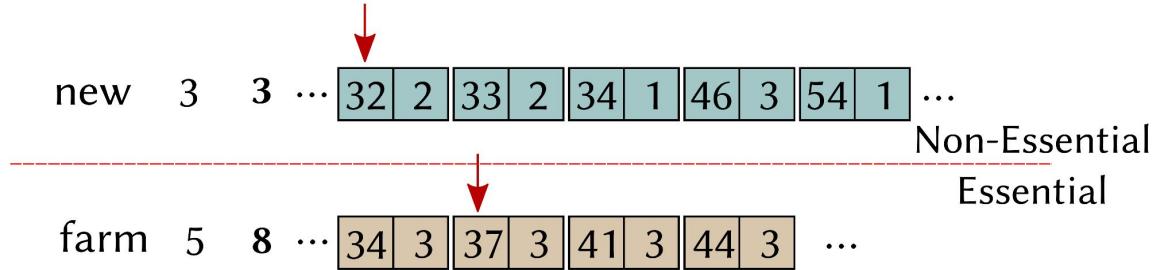


Top- k documents (min-heap) ($k = 2$)

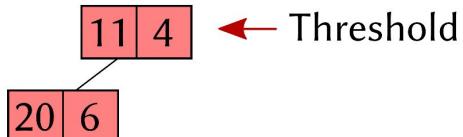


MaxScore: No Sorting Required!

Term U_t PS Postings List



Top- k documents (min-heap) ($k = 2$)



Always draw the next candidate document from the set of *essential* lists, accessing the non-essential lists only when computing a full document score.

WAND

Algorithm 3.3: WAND processing.

Input : An array \mathcal{P} of n postings cursors, the largest document identifier, MaxDocID, and the number of desired results, k .

Output : The top- k documents.

```

1 Heap ← {}
2 θ ← 0
3 SortByDocID(ℳ)
4 while true do
5   | UpperBound ← 0
6   | PivotList ← 0
7   | PivotID ← MaxDocID
8   | while PivotList < n do      // Find the pivot list and pivot document.
9   |   | UpperBound ← UpperBound + ℳ[PivotList].upperbound()
10  |   | if UpperBound > θ then
11  |   |   | PivotID ← ℳ[PivotList].docid()
12  |   |   | while PivotList+1 < n and ℳ[PivotList+1].docid() = PivotID do
13  |   |   |   | PivotList ← PivotList + 1
14  |   |   | end
15  |   |   | break
16  |   | end
17  |   | PivotList ← PivotList + 1
18 end
19 if UpperBound ≤ θ or PivotID = MaxDocID then      // No pivot. Exit.
20   | break
21 end
22 if ℳ[0].docid() = PivotID then      // Can evaluate the pivot document.
23   | Score ← 0
24   | for List ← 0 to PivotList do
25   |   | Score ← Score + ℳ[List].score()
26   |   | ℳ[List].next()
27 end
28 Heaps.push((PivotID, Score))
29 θ ← Heaps.min()
30 SortByDocID(ℳ)
31 end
32 else                                     // Need to align lists with the pivot.
33   | while ℳ[PivotList] = PivotID do
34   |   | PivotList ← PivotList - 1
35   | end
36   | ℳ[PivotList].nextGEQ(PivotID)
37   | BubbleDown(ℳ, PivotList)
38 end
39 end
40 return Heap

```

WAND

Algorithm 3.3: WAND processing.

Input : An array \mathcal{P} of n postings cursors, the largest document identifier, MaxDocID, and the number of desired results, k .

Output : The top- k documents.

```

1 Heap ← {}
2 θ ← 0
3 SortByDocID(ℳ)
4 while true do
5   UpperBound ← 0
6   PivotList ← 0
7   PivotID ← MaxDocID
8   while PivotList <  $n$  do          // Find the pivot list and pivot document.
9     | UpperBound ← UpperBound + ℳ[PivotList].upperbound()
10    | if UpperBound > θ then
11      |   | PivotID ← ℳ[PivotList].docid()
12      |   | while PivotList+1 <  $n$  and ℳ[PivotList+1].docid() = PivotID do
13      |   |   | PivotList ← PivotList + 1
14      |   | end
15      |   | break
16    | end
17    | PivotList ← PivotList + 1
18  end
19  if UpperBound ≤ θ or PivotID = MaxDocID then           // No pivot. Exit.
20    | break
21  end
22  if ℳ[0].docid() = PivotID then             // Can evaluate the pivot document.
23    | Score ← 0
24    | for List ← 0 to PivotList do
25    |   | Score ← Score + ℳ[List].score()
26    |   | ℳ[List].next()
27    | end
28    | Heaps.push((PivotID, Score))
29    | θ ← Heaps.min()
30    | SortByDocID(ℳ)
31  end
32  else                                         // Need to align lists with the pivot.
33    | while ℳ[PivotList] = PivotID do
34    |   | PivotList ← PivotList - 1
35    | end
36    | ℳ[PivotList].nextGEQ(PivotID)
37    | BubbleDown(ℳ, PivotList)
38  end
39 end
40 return Heap

```

Find the first document that *might* enter the top- k based on the upper-bound estimations

This is called the “pivot” document.

We also track which lists are “in play”.

WAND

Algorithm 3.3: WAND processing.

Input : An array \mathcal{P} of n postings cursors, the largest document identifier, MaxDocID, and the number of desired results, k .

Output : The top- k documents.

```

1 Heap ← {}
2 θ ← 0
3 SortByDocID(ℳ)
4 while true do
5   | UpperBound ← 0
6   | PivotList ← 0
7   | PivotID ← MaxDocID
8   | while PivotList < n do      // Find the pivot list and pivot document.
9   |   | UpperBound ← UpperBound + ℳ[PivotList].upperbound()
10  |   | if UpperBound > θ then
11  |   |   | PivotID ← ℳ[PivotList].docid()
12  |   |   | while PivotList+1 < n and ℳ[PivotList+1].docid() = PivotID do
13  |   |   |   | PivotList ← PivotList + 1
14  |   |   | end
15  |   |   | break
16  |   | end
17  |   | PivotList ← PivotList + 1
18 ena
19 if UpperBound ≤ θ or PivotID = MaxDocID then      // No pivot. Exit.
20   | break
21 end
22 if ℳ[0].docid() = PivotID then      // Can evaluate the pivot document.
23   | Score ← 0
24   | for List ← 0 to PivotList do
25   |   | Score ← Score + ℳ[List].score()
26   |   | ℳ[List].next()
27   | end
28   | Heap.push((PivotID, Score))
29   | θ ← Heap.min()
30   | SortByDocID(ℳ)
31 end
32 else
33   | | // Need to align lists with the pivot.
34   | | while ℳ[PivotList] = PivotID do
35   |   | | PivotList ← PivotList - 1
36   |   | | ℳ[PivotList].nextGEQ(PivotID)
37   |   | | BubbleDown(ℳ, PivotList)
38   | end
39 end
40 return Heap

```

We didn't find a pivot - we are done!

WAND

Algorithm 3.3: WAND processing.

Input : An array \mathcal{P} of n postings cursors, the largest document identifier, MaxDocID, and the number of desired results, k .

Output : The top- k documents.

```

1 Heap ← {}
2 θ ← 0
3 SortByDocID(ℳ)
4 while true do
5   | UpperBound ← 0
6   | PivotList ← 0
7   | PivotID ← MaxDocID
8   | while PivotList < n do      // Find the pivot list and pivot document.
9   |   | UpperBound ← UpperBound + ℳ[PivotList].upperbound()
10  |   if UpperBound > θ then
11    |     | PivotID ← ℳ[PivotList].docid()
12    |     | while PivotList+1 < n and ℳ[PivotList+1].docid() = PivotID do
13    |     |     | PivotList ← PivotList + 1
14    |     |   end
15    |     |   break
16    |   end
17    |   PivotList ← PivotList + 1
18  end
19  if UpperBound ≤ θ or PivotID = MaxDocID then      // No pivot. Exit.
20    | break
21  end
22  if ℳ[0].docid() = PivotID then          // Can evaluate the pivot document.
23    | Score ← 0
24    | for List ← 0 to PivotList do
25    |   | Score ← Score + ℳ[List].score()
26    |   | ℳ[List].next()
27    | end
28    | Heaps.push((PivotID, Score))
29    | θ ← Heaps.min()
30    | SortByDocID(ℳ)
31  end
32  else                                     // Need to align lists with the pivot.
33    | while ℳ[PivotList] = PivotID do
34    |   | PivotList ← PivotList - 1
35    | end
36    | ℳ[PivotList].nextGEQ(PivotID)
37    | BubbleDown(ℳ, PivotList)
38  end
39 end
40 return Heap

```

If the first list “points to” the pivot document, then all lists “in play” are pointing to the pivot. We score the pivot, try to add it to the heap, and then re-sort the cursors.

WAND

Algorithm 3.3: WAND processing.

Input : An array \mathcal{P} of n postings cursors, the largest document identifier, MaxDocID, and the number of desired results, k .

Output : The top- k documents.

```

1 Heap ← {}
2 θ ← 0
3 SortByDocID( $\mathcal{P}$ )
4 while true do
5   UpperBound ← 0
6   PivotList ← 0
7   PivotID ← MaxDocID
8   while PivotList < n do      // Find the pivot list and pivot document.
9     | UpperBound ← UpperBound +  $\mathcal{P}[\text{PivotList}].upperbound()$ 
10    | if UpperBound > θ then
11      |   | PivotID ←  $\mathcal{P}[\text{PivotList}].docid()$ 
12      |   | while PivotList+1 < n and  $\mathcal{P}[\text{PivotList}+1].docid() = \text{PivotID}$  do
13      |   |   | PivotList ← PivotList + 1
14      |   | end
15      |   | break
16    | end
17    | PivotList ← PivotList + 1
18  end
19  if UpperBound ≤ θ or PivotID = MaxDocID then      // No pivot. Exit.
20    | break
21  end
22  if  $\mathcal{P}[0].docid()$  = PivotID then      // Can evaluate the pivot document.
23    | Score ← 0
24    | for List ← 0 to PivotList do
25    |   | Score ← Score +  $\mathcal{P}[\text{List}].score()$ 
26    |   |  $\mathcal{P}[\text{List}].next()$ 
27    | end
28    |  $\text{Heap.push}(\text{PivotID}, \text{Score})$ 
29    | θ ← Heap.min()
30    | SortByDocID( $\mathcal{P}$ )
31  end
32  else          // Need to align lists with the pivot.
33    | while  $\mathcal{P}[\text{PivotList}] = \text{PivotID}$  do
34    |   | PivotList ← PivotList - 1
35    | end
36    |  $\mathcal{P}[\text{PivotList}].nextGEQ(\text{PivotID})$ 
37    | BubbleDown( $\mathcal{P}$ , PivotList)
38  end
39 end
40 return Heap

```

Otherwise, there are lists that point to documents smaller than the pivot. We need to move them up to the pivot before we score. Note that this may require partial sorting (BubbleDown)

MaxScore

Algorithm 3.2: DAAT MaxScore processing.

Input : An array \mathcal{P} of n postings cursors which are sorted increasing on their upper-bound values, the largest document identifier, MaxDocID, and the number of desired results, k .

Output : The top- k documents.

```
1 Heap ← {}
2 θ ← 0
3 CumulativeBounds ← {}
4 CumulativeBounds[0] ←  $\mathcal{P}[0].upperbound()$ 
5 for i ← 1 to  $n - 1$  do
6   | CumulativeBounds[i] ← CumulativeBounds[i - 1] +  $\mathcal{P}[i].upperbound()$ 
7 end
8 PivotList ← 0
9 PivotID ← MinimumDocID( $\mathcal{P}$ )
10 while PivotID ≤ MaxDocID and PivotList < n do
11   | Score ← 0
12   | NextCandidate ← MaxDocID
13   | for i ← PivotList to  $n - 1$  do          // Score essential lists.
14     |   if  $\mathcal{P}[i].docid()$  = PivotID then
15       |     | Score ← Score +  $\mathcal{P}[i].score()$ 
16       |     |  $\mathcal{P}[i].next()$ 
17     |   end
18     |   if  $\mathcal{P}[i].docid()$  < NextCandidate then
19       |     | NextCandidate ←  $\mathcal{P}[i].docid()$ 
20     |   end
21   | end
22   | for i ← PivotList - 1 to 0 do // Complete scoring on non-essential lists.
23     |   if Score + CumulativeBounds[i] ≤ θ then
24       |     | break
25     |   end
26     |    $\mathcal{P}[i].nextGEQ(PivotID)$ 
27     |   if  $\mathcal{P}[i].docid()$  = PivotID then
28       |     | Score ← Score +  $\mathcal{P}[i].score()$ 
29     |   end
30   | end
31   | CurrentBound ← θ
32   | Heap.push((PivotID, Score))
33   | θ ← Heap.min()
34   | if CurrentBound < θ then           // The heap threshold increased.
35     |   | while PivotList < n and CumulativeBounds[PivotList] ≤ θ do
36     |     |   | PivotList ← PivotList + 1
37     |   | end
38   | end
39   | PivotID ← NextCandidate
40 end
41 return Heap
```

MaxScore

Algorithm 3.2: DAAT MaxScore processing.

```
Input : An array  $\mathcal{P}$  of  $n$  postings cursors which are sorted increasing on their upper-bound values, the largest document identifier, MaxDocID, and the number of desired results,  $k$ .  
Output : The top- $k$  documents.  
1 Heap  $\leftarrow \{\}$   
2  $\theta \leftarrow 0$   
3 CumulativeBounds  $\leftarrow \{\}$   
4 CumulativeBounds[0]  $\leftarrow \mathcal{P}[0].upperbound()$   
5 for  $i \leftarrow 1$  to  $n - 1$  do  
6   | CumulativeBounds[i]  $\leftarrow$  CumulativeBounds[i-1] +  $\mathcal{P}[i].upperbound()$   
7 end  
8 PivotList  $\leftarrow 0$   
9 PivotID  $\leftarrow \text{MinimumDocID}(\mathcal{P})$   
10 while PivotID  $\leq$  MaxDocID and PivotList  $< n$  do  
11   | Score  $\leftarrow 0$   
12   | NextCandidate  $\leftarrow$  MaxDocID  
13   | for  $i \leftarrow$  PivotList to  $n - 1$  do // Score essential lists.  
14     | | if  $\mathcal{P}[i].docid() =$  PivotID then  
15       | | | Score  $\leftarrow$  Score +  $\mathcal{P}[i].score()$   
16       | | |  $\mathcal{P}[i].next()$   
17     | | end  
18     | | if  $\mathcal{P}[i].docid() <$  NextCandidate then  
19       | | | NextCandidate  $\leftarrow \mathcal{P}[i].docid()$   
20     | | end  
21   | end  
22   | for  $i \leftarrow$  PivotList - 1 to 0 do // Complete scoring on non-essential lists.  
23     | | if Score + CumulativeBounds[i]  $\leq \theta$  then  
24       | | | break  
25     | | end  
26     | |  $\mathcal{P}[i].nextGEQ(\text{PivotID})$   
27     | | if  $\mathcal{P}[i].docid() =$  PivotID then  
28       | | | Score  $\leftarrow$  Score +  $\mathcal{P}[i].score()$   
29     | | end  
30   | end  
31   | CurrentBound  $\leftarrow \theta$   
32   | Heap.push((PivotID, Score))  
33   |  $\theta \leftarrow \text{Heap}.min()$   
34   | if CurrentBound  $< \theta$  then // The heap threshold increased.  
35     | | while PivotList  $< n$  and CumulativeBounds[PivotList]  $\leq \theta$  do  
36       | | | PivotList  $\leftarrow$  PivotList + 1  
37     | | end  
38   | end  
39   | PivotID  $\leftarrow$  NextCandidate  
40 end  
41 return Heap
```

Compute the cumulative sum of the upper-bounds

MaxScore

Algorithm 3.2: DAAT MaxScore processing.

```

Input : An array  $\mathcal{P}$  of  $n$  postings cursors which are sorted increasing on their
         upper-bound values, the largest document identifier, MaxDocID, and the
         number of desired results,  $k$ .
Output : The top- $k$  documents.

1 Heap  $\leftarrow \{\}$ 
2  $\theta \leftarrow 0$ 
3 CumulativeBounds  $\leftarrow \{\}$ 
4 CumulativeBounds[0]  $\leftarrow \mathcal{P}[0].upperbound()$ 
5 for  $i \leftarrow 1$  to  $n - 1$  do
6   | CumulativeBounds[i]  $\leftarrow$  CumulativeBounds[i-1] +  $\mathcal{P}[i].upperbound()$ 
7 end
8 PivotList  $\leftarrow 0$ 
9 PivotID  $\leftarrow \text{MinimumDocID}(\mathcal{P})$ 
10 while PivotID  $\leq$  MaxDocID and PivotList  $< n$  do
11   | Score  $\leftarrow 0$ 
12   | NextCandidate  $\leftarrow \text{MaxDocID}$ 
13   | for  $i \leftarrow \text{PivotList}$  to  $n - 1$  do
14     |   if  $\mathcal{P}[i].docid() = \text{PivotID}$  then
15       |     | Score  $\leftarrow$  Score +  $\mathcal{P}[i].score()$ 
16       |     |  $\mathcal{P}[i].next()$ 
17     |   end
18     |   if  $\mathcal{P}[i].docid() < \text{NextCandidate}$  then
19       |     | NextCandidate  $\leftarrow \mathcal{P}[i].docid()$ 
20     |   end
21   | end
22   | for  $i \leftarrow \text{PivotList} - 1$  to 0 do // complete scoring on non-essential lists.
23     |   if Score + CumulativeBounds[i]  $\leq \theta$  then
24       |     | break
25     |   end
26     |    $\mathcal{P}[i].nextGEQ(\text{PivotID})$ 
27     |   if  $\mathcal{P}[i].docid() = \text{PivotID}$  then
28       |     | Score  $\leftarrow$  Score +  $\mathcal{P}[i].score()$ 
29     |   end
30   | end
31   | CurrentBound  $\leftarrow \theta$ 
32   | Heap.push((PivotID, Score))
33   |  $\theta \leftarrow \text{Heap}.min()$ 
34   | if CurrentBound  $< \theta$  then           // The heap threshold increased.
35     |   | while PivotList  $< n$  and CumulativeBounds[PivotList]  $\leq \theta$  do
36     |   |   | PivotList  $\leftarrow$  PivotList + 1
37     |   | end
38   | end
39   | PivotID  $\leftarrow$  NextCandidate
40 end
41 return Heap

```

The pivot document is the minimum document in the essential lists. We score this document, and also track the next pivot candidate.

MaxScore

Algorithm 3.2: DAAT MaxScore processing.

```
Input : An array  $\mathcal{P}$  of  $n$  postings cursors which are sorted increasing on their upper-bound values, the largest document identifier, MaxDocID, and the number of desired results,  $k$ .  
Output : The top- $k$  documents.  
1 Heap  $\leftarrow \{\}$   
2  $\theta \leftarrow 0$   
3 CumulativeBounds  $\leftarrow \{\}$   
4 CumulativeBounds[0]  $\leftarrow \mathcal{P}[0].upperbound()$   
5 for  $i \leftarrow 1$  to  $n - 1$  do  
6   | CumulativeBounds[i]  $\leftarrow$  CumulativeBounds[i-1] +  $\mathcal{P}[i].upperbound()$   
7 end  
8 PivotList  $\leftarrow 0$   
9 PivotID  $\leftarrow \text{MinimumDocID}(\mathcal{P})$   
10 while PivotID  $\leq$  MaxDocID and PivotList  $< n$  do  
11   | Score  $\leftarrow 0$   
12   | NextCandidate  $\leftarrow \text{MaxDocID}$   
13   | for  $i \leftarrow \text{PivotList}$  to  $n - 1$  do // Score essential lists.  
14     |   | if  $\mathcal{P}[i].docid() = \text{PivotID}$  then  
15       |   |   | Score  $\leftarrow$  Score +  $\mathcal{P}[i].score()$   
16       |   |   |  $\mathcal{P}[i].next()$   
17     |   | end  
18     |   | if  $\mathcal{P}[i].docid() < \text{NextCandidate}$  then  
19       |   |   | NextCandidate  $\leftarrow \mathcal{P}[i].docid()$   
20     |   | end  
21   | end  
22   | for  $i \leftarrow \text{PivotList} - 1$  to 0 do // Complete scoring on non-essential lists.  
23     |   | if Score + CumulativeBounds[i]  $\leq \theta$  then  
24       |   |   | break  
25     |   | end  
26     |   |  $\mathcal{P}[i].nextGEQ(\text{PivotID})$   
27     |   | if  $\mathcal{P}[i].docid() = \text{PivotID}$  then  
28       |   |   | Score  $\leftarrow$  Score +  $\mathcal{P}[i].score()$   
29     |   | end  
30   | end  
31   | CurrentBound  $\leftarrow \theta$   
32   | Heap.push((PivotID, Score))  
33   |  $\theta \leftarrow \text{Heap}.min()$   
34   | if CurrentBound  $< \theta$  then // The heap threshold increased.  
35     |   | while PivotList  $< n$  and CumulativeBounds[PivotList]  $\leq \theta$  do  
36       |   |   | PivotList  $\leftarrow$  PivotList + 1  
37     |   | end  
38   | end  
39   | PivotID  $\leftarrow$  NextCandidate  
40 end  
41 return Heap
```

The pivot document may also occur in the non-essential lists; we complete scoring the pivot here.

MaxScore

Algorithm 3.2: DAAT MaxScore processing.

```

Input : An array  $\mathcal{P}$  of  $n$  postings cursors which are sorted increasing on their
         upper-bound values, the largest document identifier, MaxDocID, and the
         number of desired results,  $k$ .
Output : The top- $k$  documents.

1 Heap  $\leftarrow \{\}$ 
2  $\theta \leftarrow 0$ 
3 CumulativeBounds  $\leftarrow \{\}$ 
4 CumulativeBounds[0]  $\leftarrow \mathcal{P}[0].upperbound()$ 
5 for  $i \leftarrow 1$  to  $n - 1$  do
6   | CumulativeBounds[i]  $\leftarrow$  CumulativeBounds[i-1] +  $\mathcal{P}[i].upperbound()$ 
7 end
8 PivotList  $\leftarrow 0$ 
9 PivotID  $\leftarrow \text{MinimumDocID}(\mathcal{P})$ 
10 while PivotID  $\leq$  MaxDocID and PivotList  $< n$  do
11   | Score  $\leftarrow 0$ 
12   | NextCandidate  $\leftarrow \text{MaxDocID}$ 
13   | for  $i \leftarrow \text{PivotList}$  to  $n - 1$  do // Score essential lists.
14     |   | if  $\mathcal{P}[i].docid() = \text{PivotID}$  then
15       |   |   | Score  $\leftarrow$  Score +  $\mathcal{P}[i].score()$ 
16       |   |   |  $\mathcal{P}[i].next()$ 
17     |   | end
18     |   | if  $\mathcal{P}[i].docid() < \text{NextCandidate}$  then
19       |   |   | NextCandidate  $\leftarrow \mathcal{P}[i].docid()$ 
20     |   | end
21   | end
22   | for  $i \leftarrow \text{PivotList} - 1$  to 0 do // Complete scoring on non-essential lists.
23     |   | if Score + CumulativeBounds[i]  $\leq \theta$  then
24       |   |   | break
25     |   | end
26     |   |  $\mathcal{P}[i].nextGEQ(\text{PivotID})$ 
27     |   | if  $\mathcal{P}[i].docid() = \text{PivotID}$  then
28       |   |   | Score  $\leftarrow$  Score +  $\mathcal{P}[i].score()$ 
29     |   | end
30   | end
31   | CurrentBound  $\leftarrow \theta$ 
32   | Heap.push((PivotID, Score))
33   |  $\theta \leftarrow \text{Heap}.min()$ 
34   | if CurrentBound  $< \theta$  then // The heap threshold increased.
35     |   | while PivotList  $< n$  and CumulativeBounds[PivotList]  $\leq \theta$  do
36       |   |   | PivotList  $\leftarrow$  PivotList + 1
37     |   | end
38   | end
39   | PivotID  $\leftarrow$  NextCandidate
40 end
41 return Heap

```

Now we check to see if the pivot can enter the heap. If so, we may need to adjust the boundary between the essential and non-essential lists.

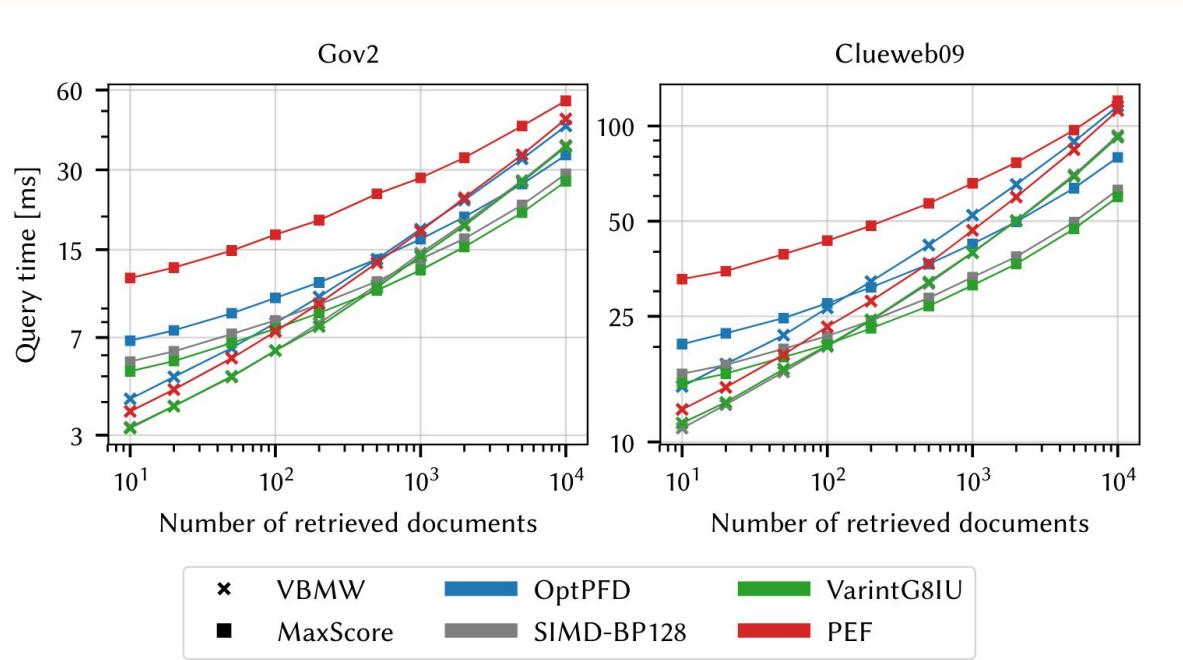
MaxScore vs WAND

- WAND typically performs well for short queries, and small values of k
 - As query length increases, the sorting operations become expensive.
 - As k increases, dynamic pruning becomes less effective, as the heap threshold is easier to beat, meaning we score more documents.

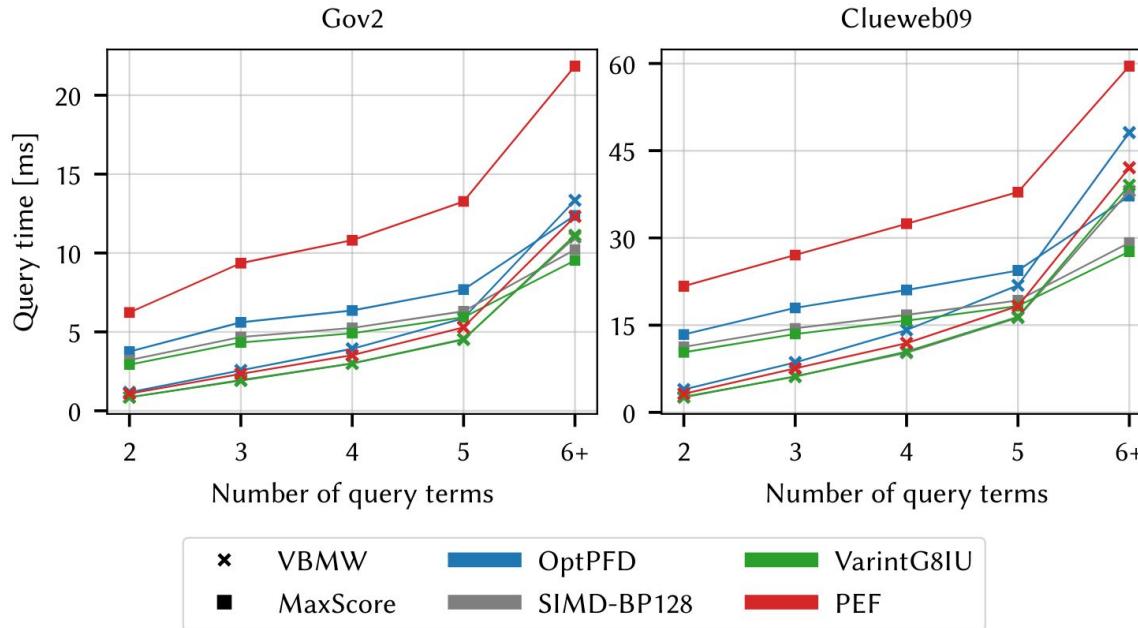
MaxScore vs WAND

- MaxScore typically performs well for long(er) queries, and large(r) values of k
 - No sorting required during processing!

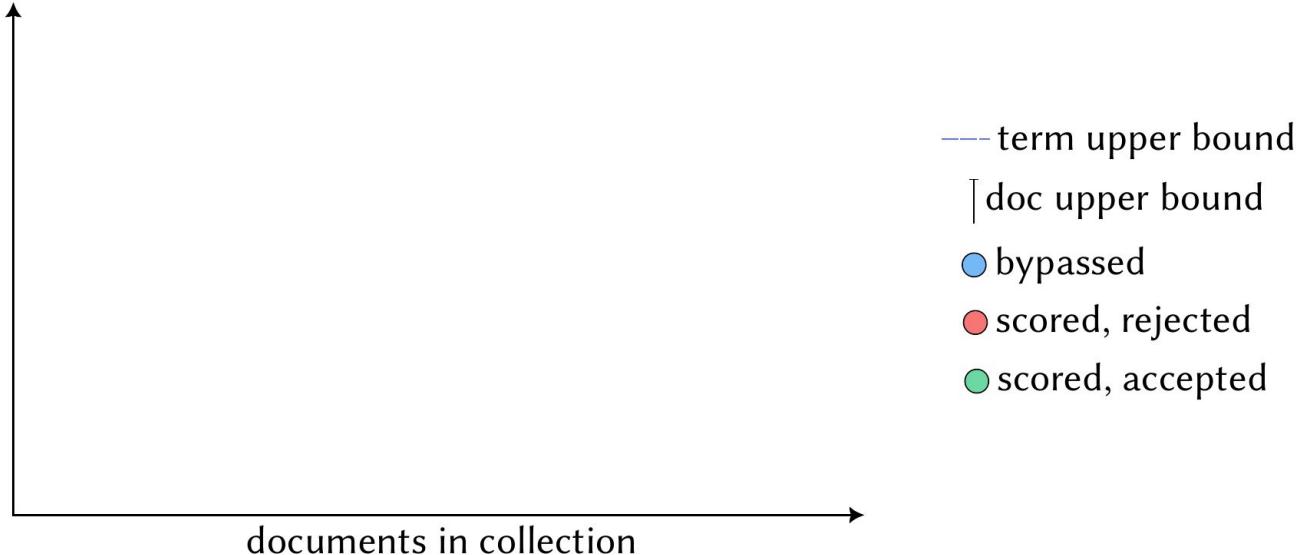
MaxScore vs WAND



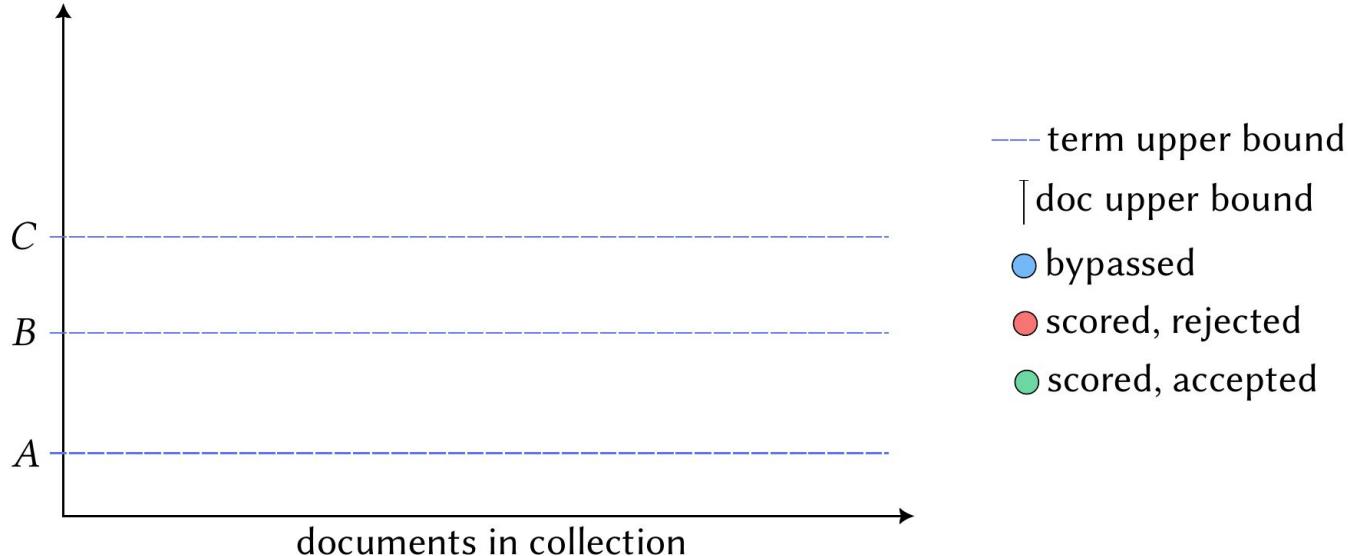
MaxScore vs WAND



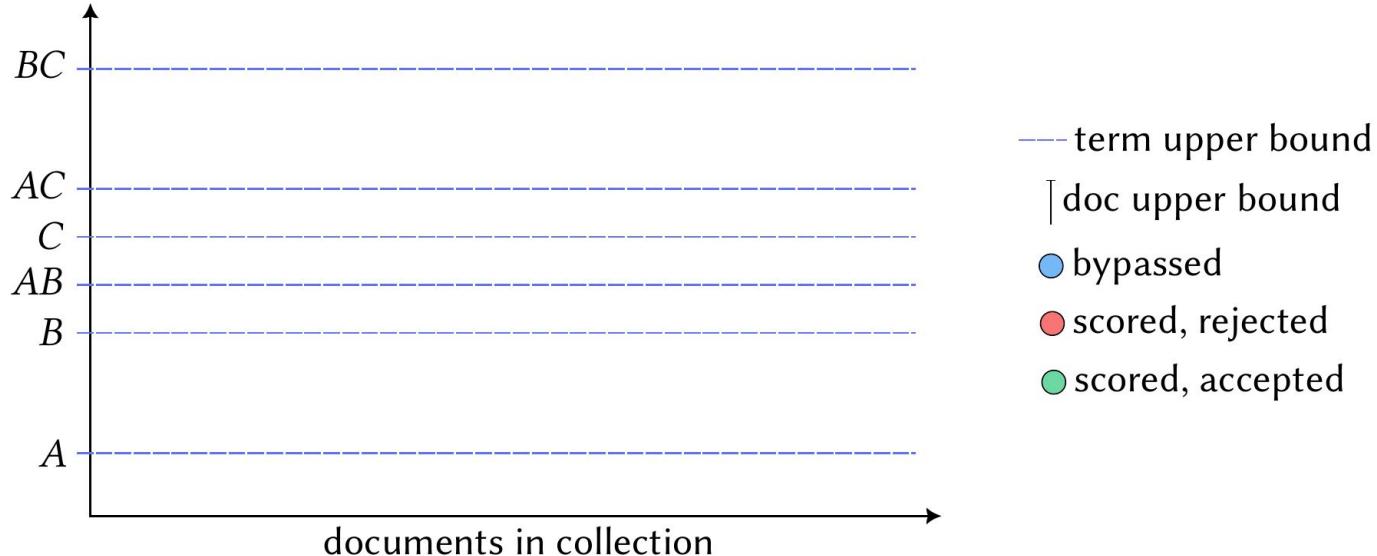
Visualizing Dynamic Pruning



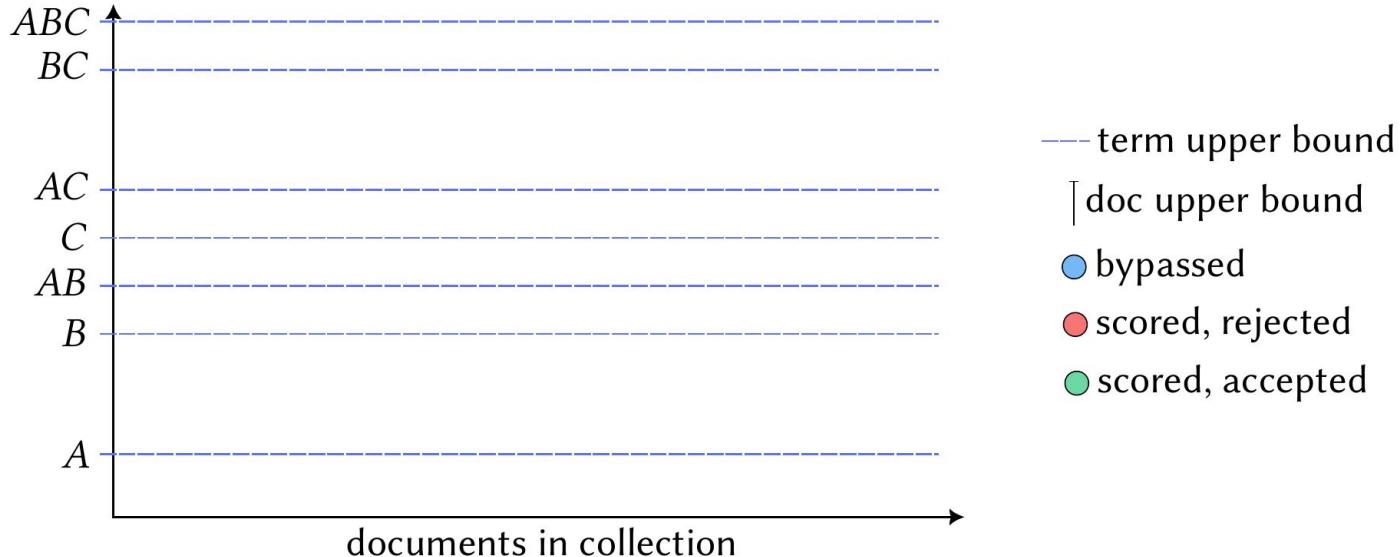
Visualizing Dynamic Pruning



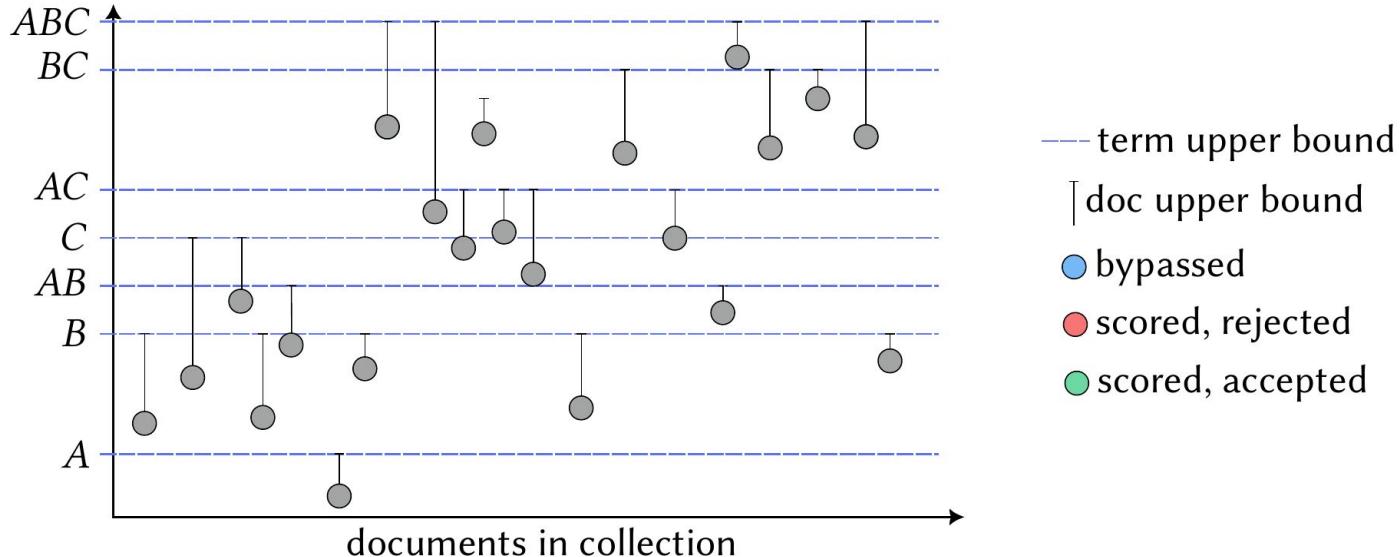
Visualizing Dynamic Pruning



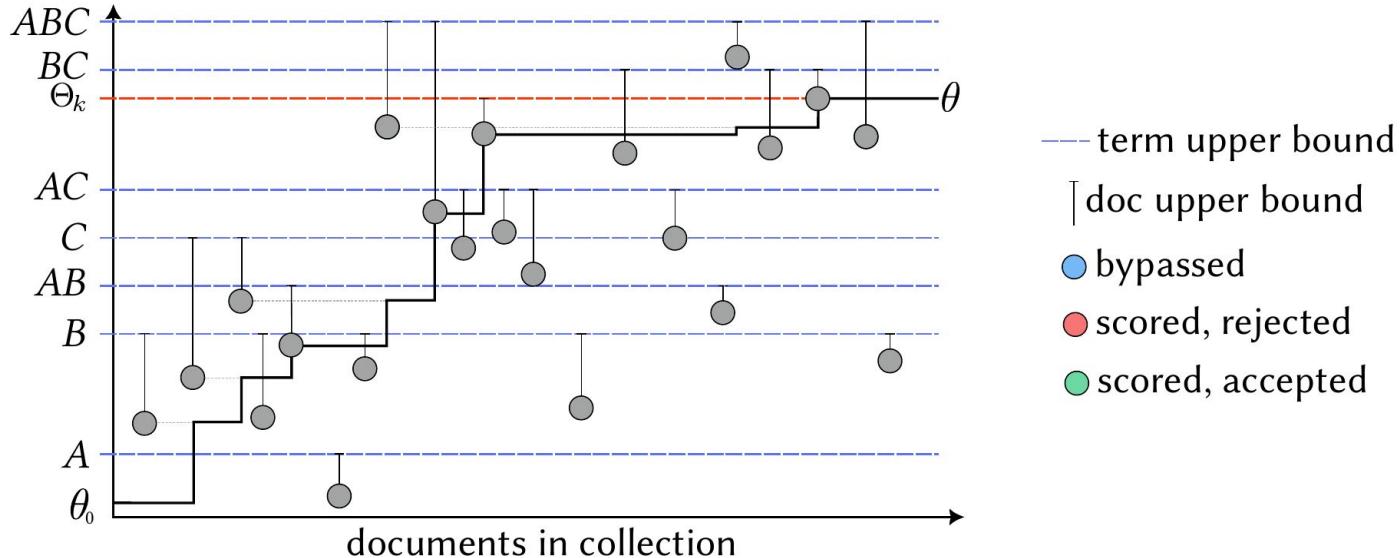
Visualizing Dynamic Pruning



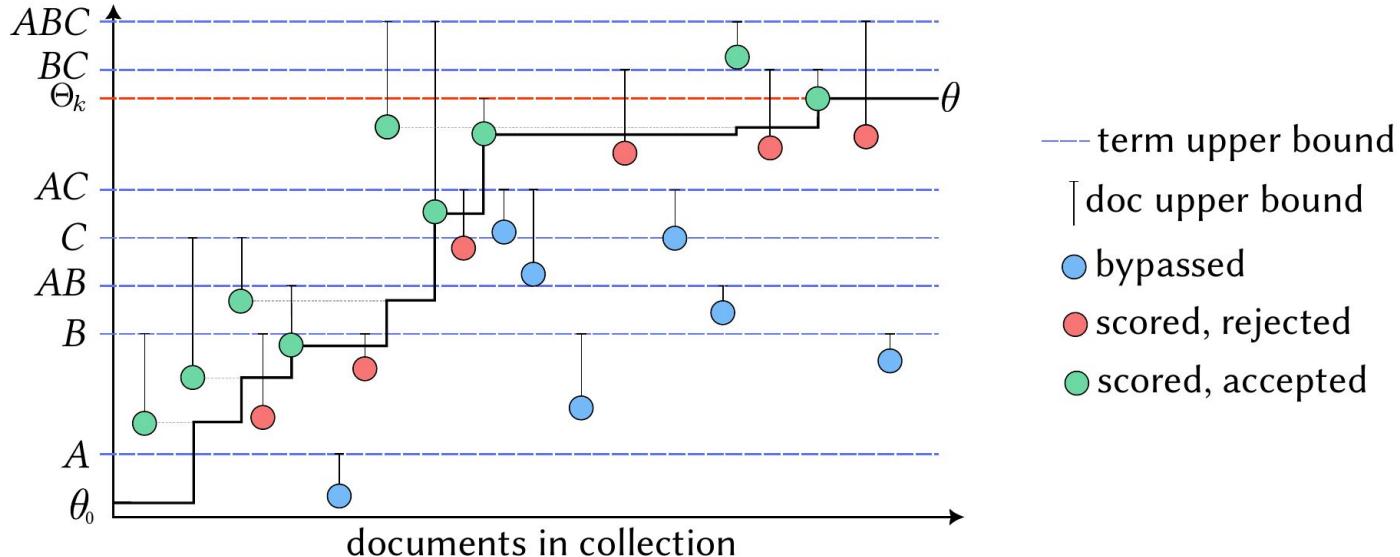
Visualizing Dynamic Pruning



Visualizing Dynamic Pruning



Visualizing Dynamic Pruning



Upper-Bound Estimation

MaxScore and WAND use list-wise upper-bounds to make estimations on document scores. These estimations are used for “go” or “no go” choices on document scoring.

But using the maximum list-wise score may not be a good estimate. How can we do better?

Block-Max Pruning Methods

One approach is to store a *per-block* upper-bound in addition to the list-wise upper-bound score.

The list-wise upper-bounds drive the initial selection of a candidate; then a localized upper-bound allows for a more accurate decision to be made before proceeding.

The obvious **downside** is the additional space consumption. But this is typically small, and these bounds can be compressed.

Block-Max Pruning Methods

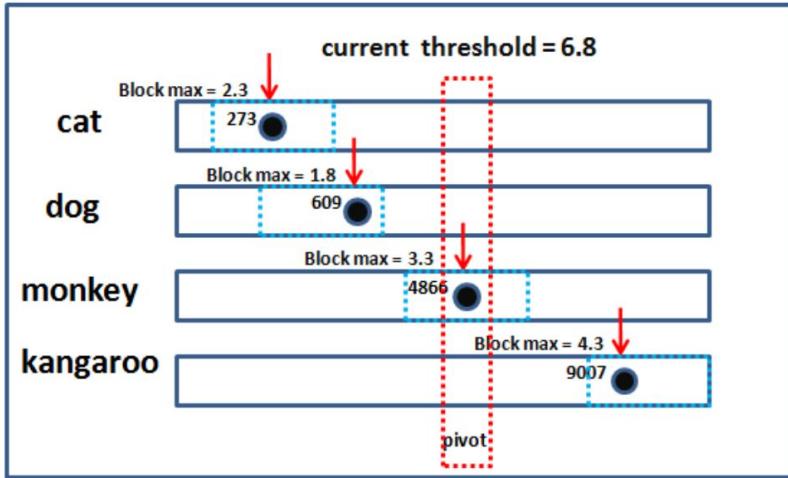


Figure 4: An example showing why directly using block max scores does not work.

We **cannot** simply use the block-max scores to decide which pivot to score, or we may skip documents that should be in the top- k .

Some optimisations intentionally do this, resulting in *unsafe* retrieval.

Block-Max Pruning Methods

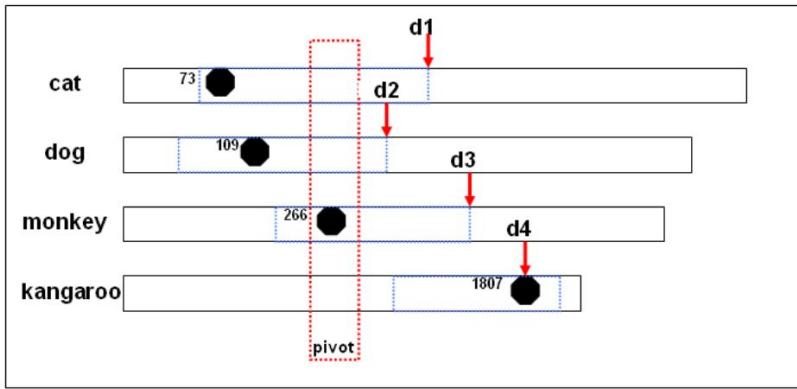


Figure 5: An example showing how *GetNewCandidate()* works. Assume 266 is the pivot and it fails to make it into the top results. In this case, we enable better skipping by choosing $\min(d_1, d_2, d_3, d_4)$ as the next possible candidate, instead of $266 + 1$

However, we **can use the block-max bounds** to make better decisions on what to process next!

In this case, the current block “configuration” cannot yield a document that will be admitted into the top- k .

Block-Max Pruning Methods

Many versions of Block-Max MaxScore and Block-Max WAND

- **Window-Based Blocks**
- **Live-Block Pruning**
- **Conditional Skipping**
- **Hybrid Approaches** (LazyBM)
- Many more...

Intuitively, all of these algorithms are variations that improve the plain BMM/BMW algorithms through specific observations; the literature is dense!

Variable-Sized Blocks

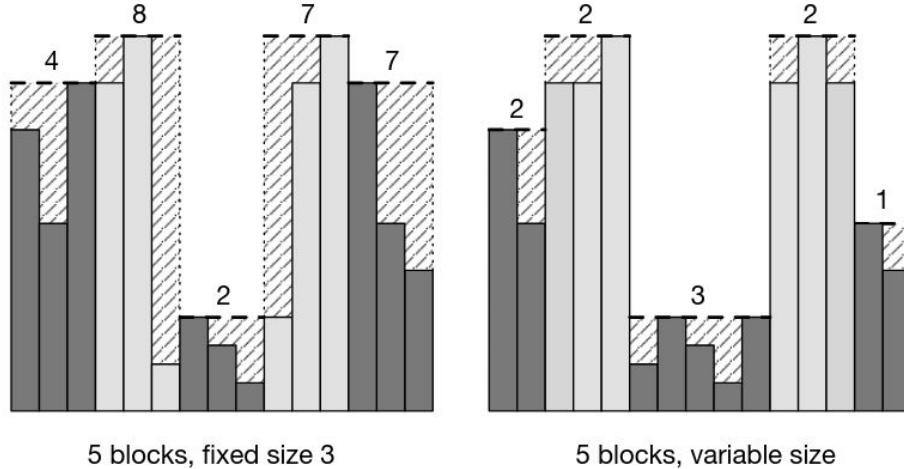


Figure 1: Block errors in constant (left) and variable (right) block partitioning.

Intuition: Fixed-size blocks may cause large within-block errors. Instead, find a variable-length partition that reduces error rate.

The Ranker does Matter!

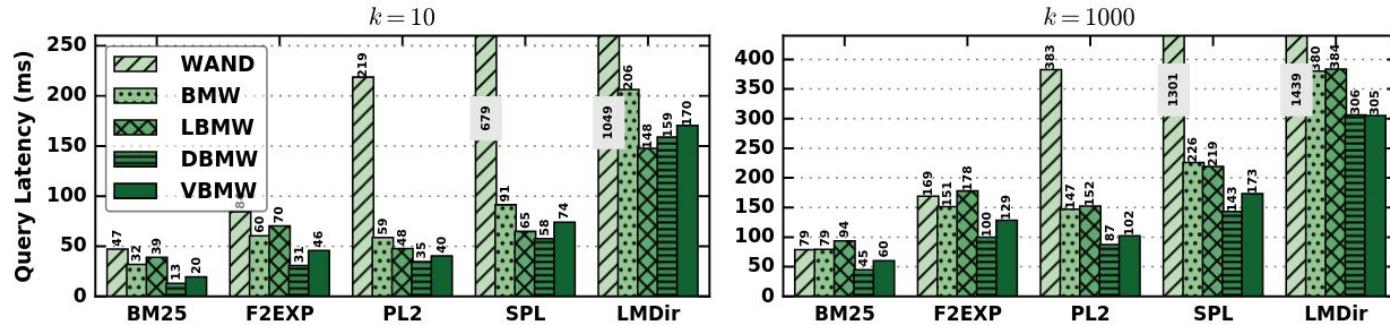


Figure 2: Mean query latency (in milliseconds) of five WAND-based strategies across five ranking models. Latency is reported for top- k queries for $k = 10$ and $k = 1000$.

The Ranker does Matter!

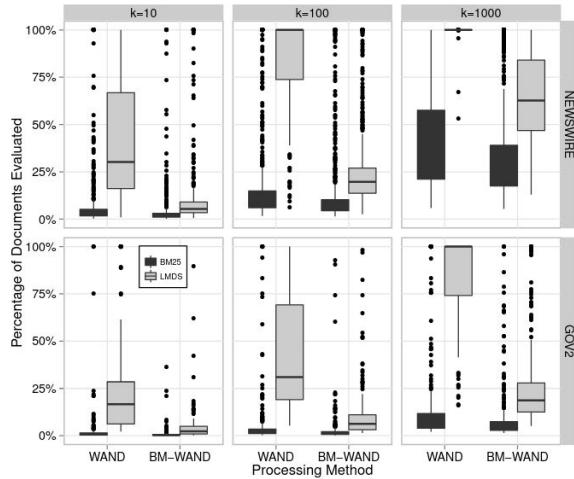


Figure 2: Number of documents scored, expressed as a fraction of the total number of documents containing any of the terms in that query. Three different retrieval depths are tested, for two different collections, for two different similarity computations, and for both WAND and BM-WAND processing.

The Ranker does Matter!

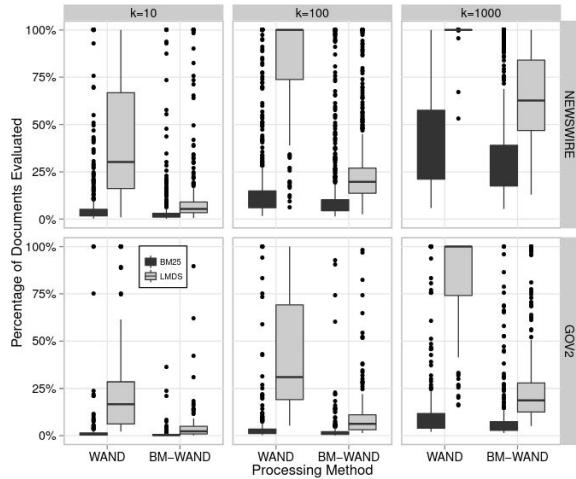


Figure 2: Number of documents scored, expressed as a fraction of the total number of documents containing any of the terms in that query. Three different retrieval depths are tested, for two different collections, for two different similarity computations, and for both WAND and BM-WAND processing.

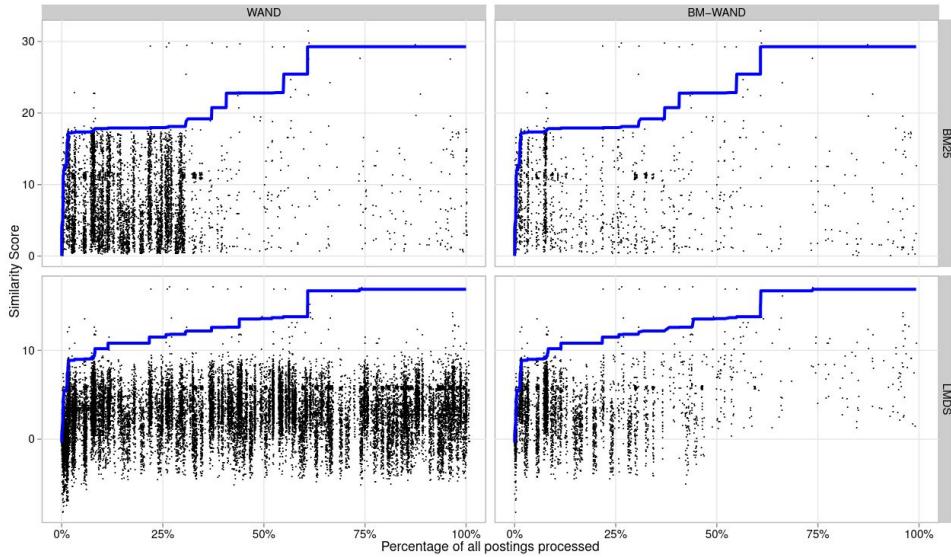
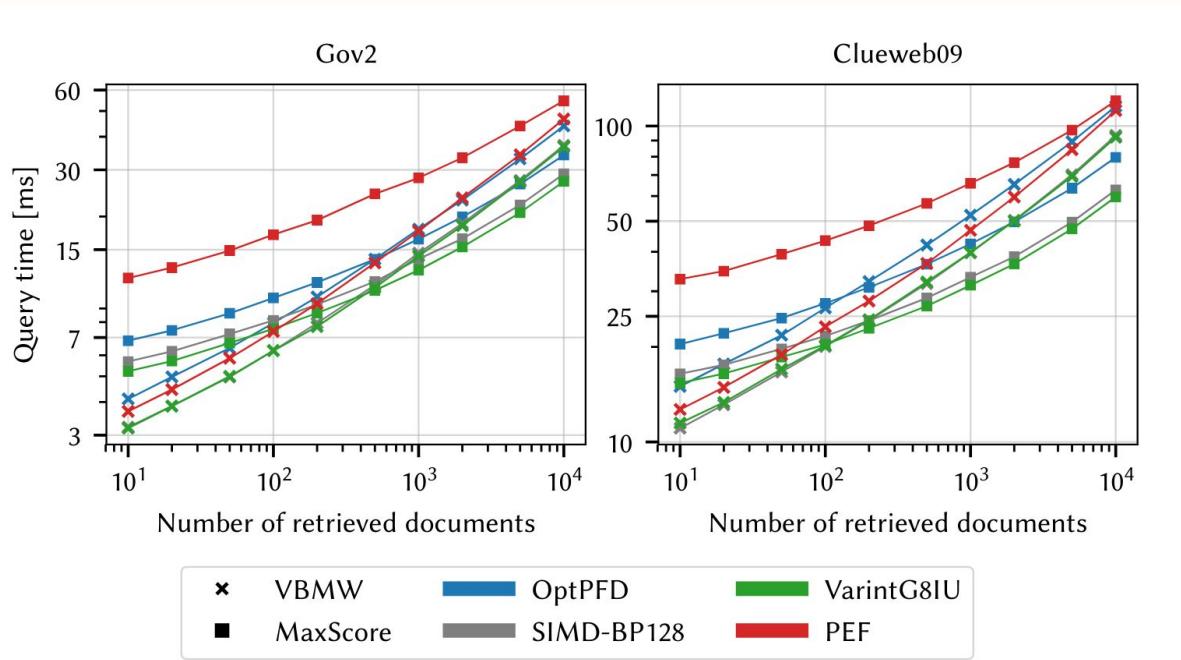
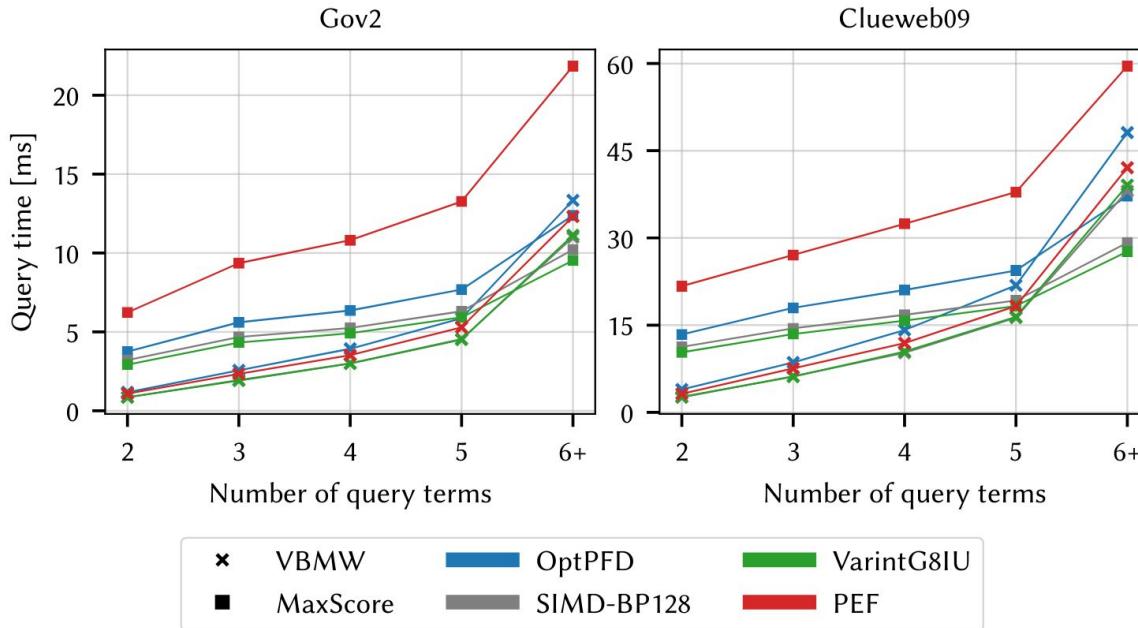


Figure 3: Distribution of evaluated documents as the postings lists for query "north_korean_counterfeiting" (topic 808) are evaluated for both WAND and BM-WAND processing using LMDS and BM25 similarity measures.

MaxScore vs VBMW

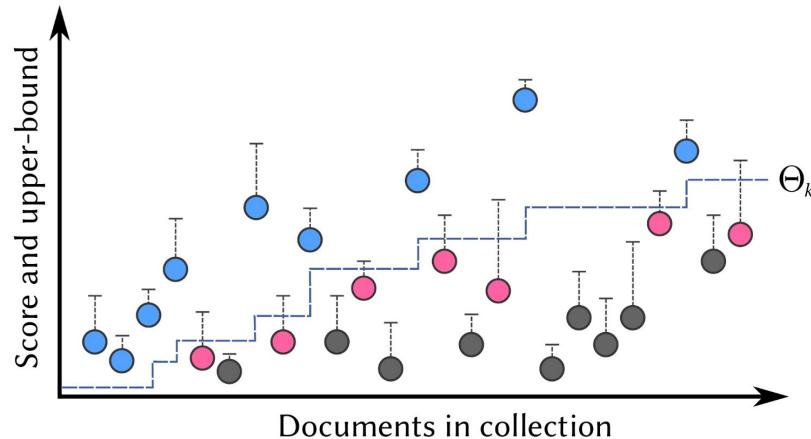


MaxScore vs VBMW



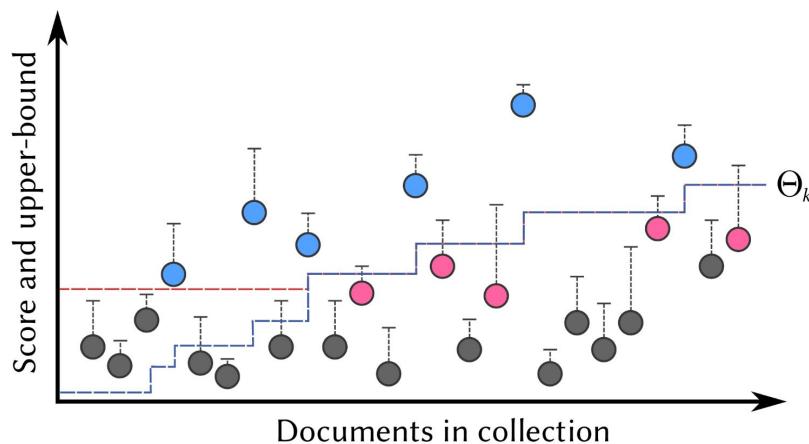
Other Enhancements

- **Threshold Priming:** If we can make a good estimate of the terminal heap threshold *before* processing, we can skip more documents!



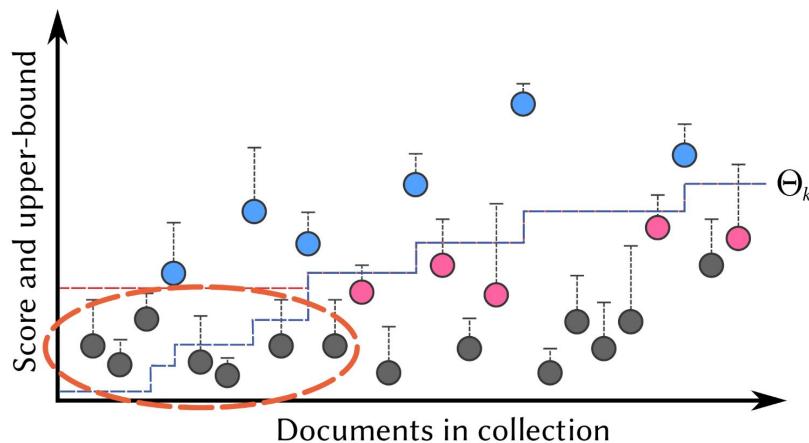
Other Enhancements

- **Threshold Priming:** If we can make a good estimate of the terminal heap threshold *before* processing, we can skip more documents!



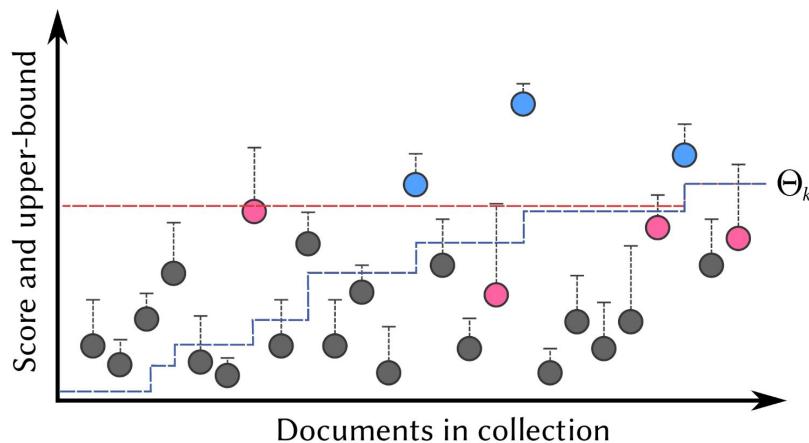
Other Enhancements

- **Threshold Priming:** If we can make a good estimate of the terminal heap threshold *before* processing, we can skip more documents!



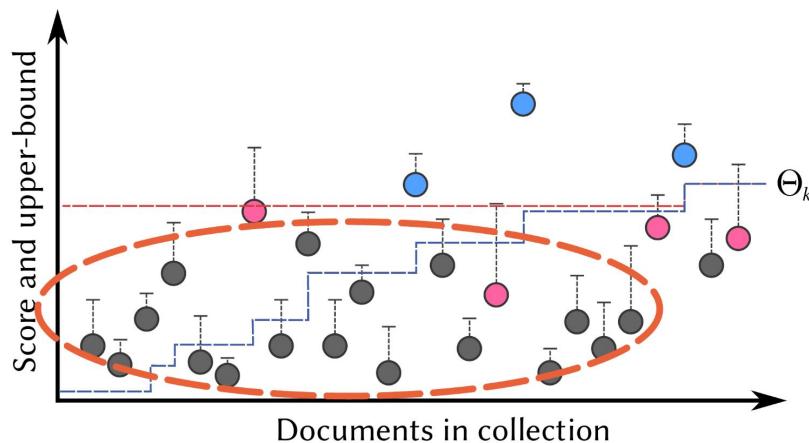
Other Enhancements

- **Threshold Priming:** If we can make a good estimate of the terminal heap threshold *before* processing, we can skip more documents!



Other Enhancements

- **Threshold Priming:** If we can make a good estimate of the terminal heap threshold *before* processing, we can skip more documents!



Other Enhancements

- **Index Reordering** (Document Identifier Reassignment)
If we can “cluster” similar documents together, we will get lots of runs of “1”s in our postings lists

Recall: Postings are strictly increasing on document id, so we *delta* code them.

Other Enhancements

- **Index Reordering** (Document Identifier Reassignment)
If we can “cluster” similar documents together, we will get lots of runs of “1”s in our postings lists

Recall: Postings are strictly increasing on document id, so we *delta* code them.

Storing Postings in Practice

DocIDs [12 | 14 | 27 | 29 | 30 | 55 | 59 | 86] ...

Storing Postings in Practice

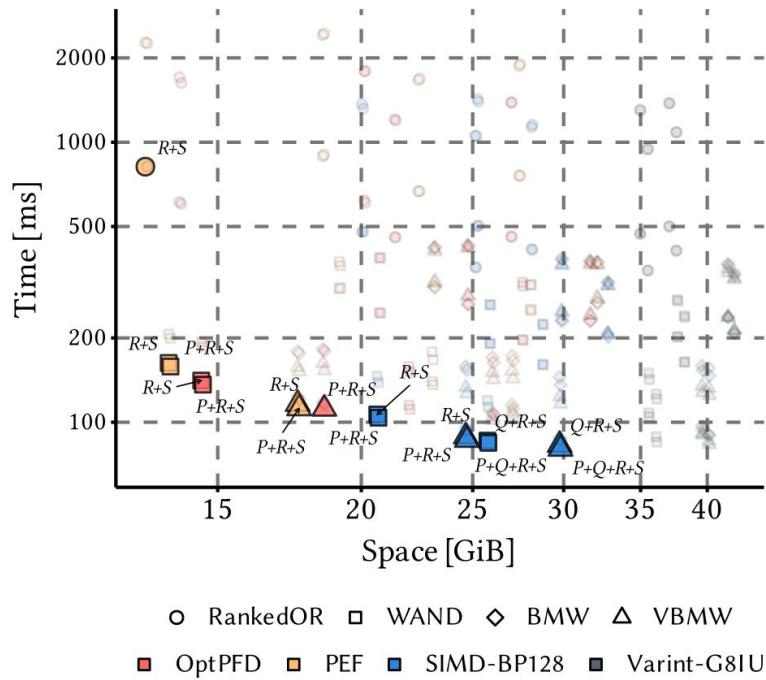
DocIDs [12 | 2 | 13 | 2 | 2 | 25 | 4 | 27] ...



Frequencies [3 | 4 | 3 | 1 | 1 | 5 | 2 | 3] ...

Frequencies [3 | 4 | 3 | 1 | 1 | 5 | 2 | 3] ...

Other Enhancements



Experiment: All combinations of **priming**, **quantization**, **reordering**, **stopping** on Clueweb12B (52 million web documents).

Efficiency innovations are, broadly speaking, additive!

Other Enhancements

| Optimizations | RankedOR | WAND | BMW | VBMW |
|---------------|-----------------------|-----------------------|-----------------------|-----------------------|
| <i>None</i> | 135.7 | 49.1 | 45.6 | 45.6 |
| <i>One</i> | 87.9 ($\times 1.5$) | 26.6 ($\times 1.8$) | 19.2 ($\times 2.4$) | 17.9 ($\times 2.6$) |
| <i>Two</i> | 62.6 ($\times 2.2$) | 20.3 ($\times 2.4$) | 14.8 ($\times 3.1$) | 13.3 ($\times 3.4$) |
| <i>Three</i> | 56.2 ($\times 2.4$) | 15.5 ($\times 3.2$) | 12.8 ($\times 3.6$) | 10.9 ($\times 4.2$) |
| <i>All</i> | 54.7 ($\times 2.5$) | 13.8 ($\times 3.6$) | 11.6 ($\times 3.9$) | 9.7 ($\times 4.7$) |

Between $2.5\times$ to $4.7\times$ speedups.

Taking stock of the current state of top- k query processing: Can retrieve the top 1000 candidates on a 52 million document collection in < 10 milliseconds.

Experiment: All combinations of **priming**, **quantization**, **reordering**, **stopping** on Clueweb12B (52 million web documents).

Efficiency innovations are, broadly speaking, additive!

Rule(s) of Thumb

1. Use **MaxScore** when **queries are long**, or **k is large**; Use **VBMW otherwise**;
2. **Stopping** is almost always a good idea;
3. Use **index reordering** if you can afford it (offline cost);
4. If you don't know which (statistical) ranker to use, just **stick to BM25** - it is fast, and well behaved;
5. Empirical experimentation is **always** beneficial!

Session I: Indexing and Retrieval

Practice

Indexing and Querying with PISA

We will now work through **Section 1** of the practical.

Tutorial: <https://shorturl.at/VExpG>

Aka: <https://github.com/pisa-engine/pisa/blob/main/test/docker/tutorial/instructions.md>

Session 1.5: Discussion & Coffee

Session II: Learned Sparse Retrieval

The key characteristic that makes inverted indexes work is sparsity.

Document 0

search is cool

Document 1

search is fun

Document 2

search is fun
for everyone

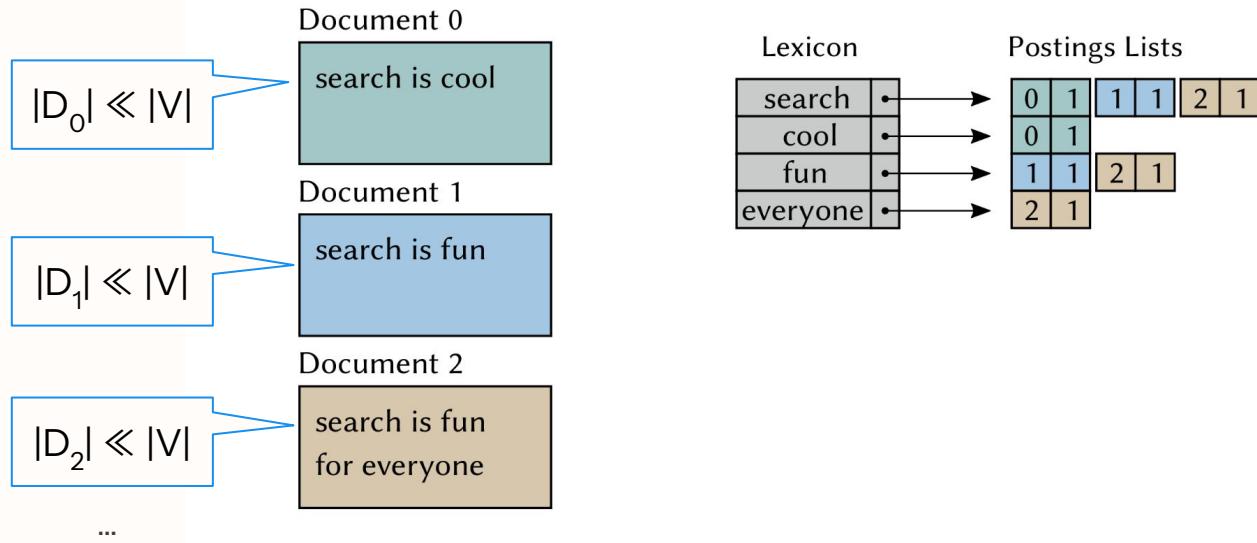
Lexicon

| | |
|----------|---|
| search | • |
| cool | • |
| fun | • |
| everyone | • |

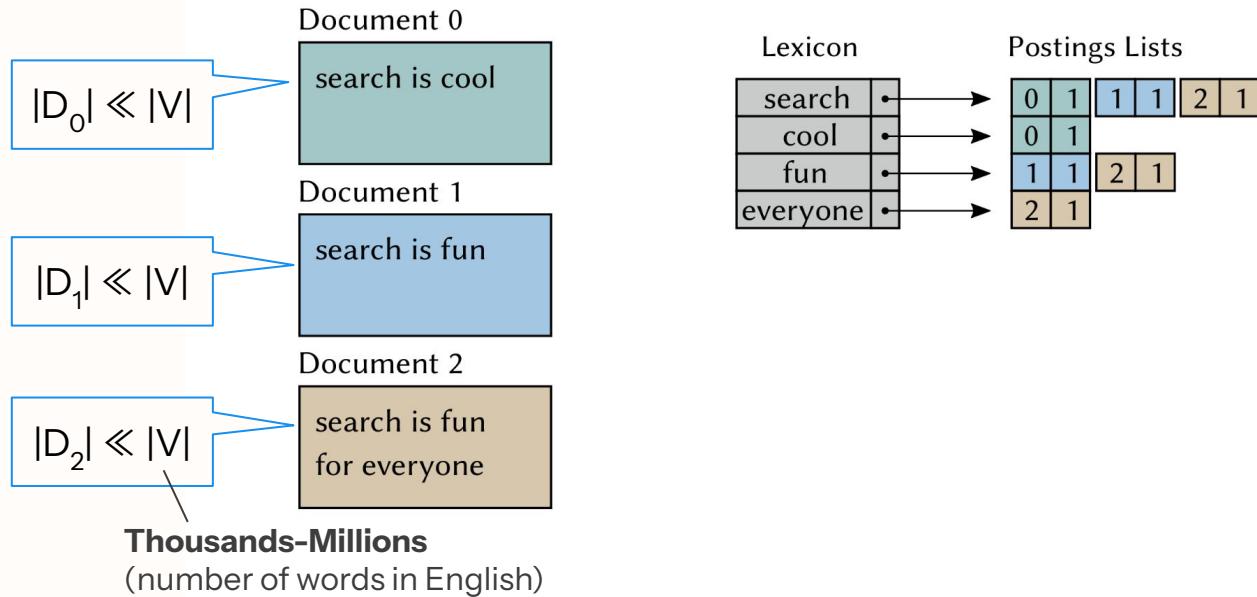
Postings Lists

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 2 | 1 |
| 0 | 1 | | | | |
| 1 | 1 | | | 2 | 1 |
| 2 | 1 | | | | |

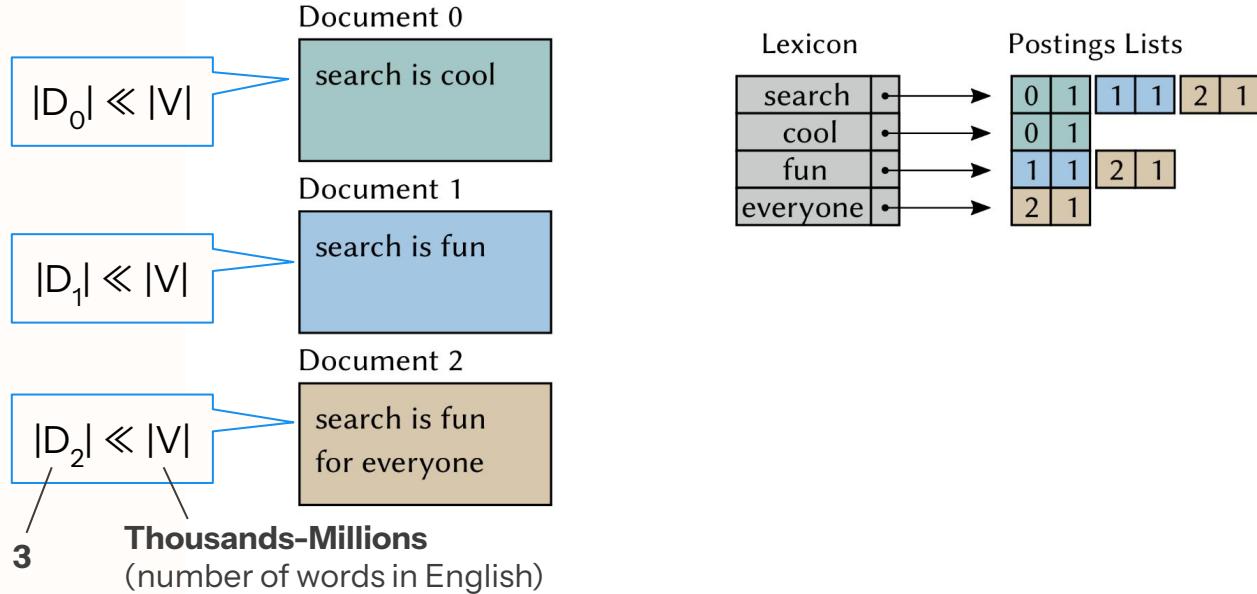
The key characteristic that makes inverted indexes work is sparsity.



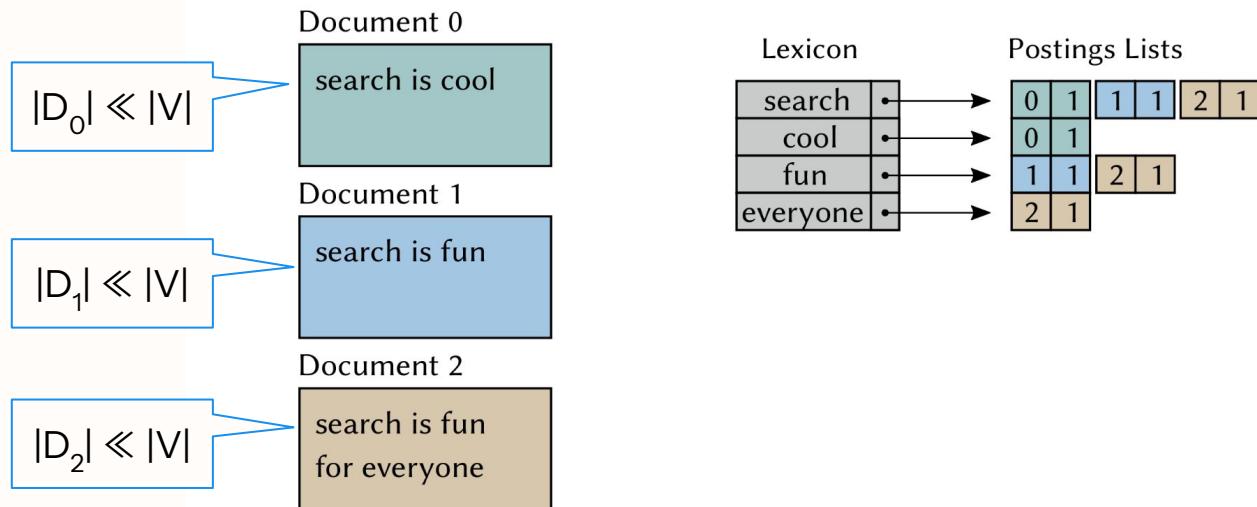
The key characteristic that makes inverted indexes work is sparsity.



The key characteristic that makes inverted indexes work is sparsity.



The key characteristic that makes inverted indexes work is sparsity.



Documents are only “about” a small number of terms

The key characteristic that makes inverted indexes work is sparsity.

Document 0
search is cool

Document 1
search is fun

Document 2
search is fun
for everyone

| | |
|----------|---|
| search | • |
| cool | • |
| fun | • |
| everyone | • |

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 2 | 1 |
| 0 | 1 | | | | |
| 1 | 1 | | | 2 | 1 |
| 2 | 1 | | | | |

3

Thousands-Billions
(number of documents in corpus)

$|P_{\text{search}}| \ll |C|$

The key characteristic that makes inverted indexes work is sparsity.

Document 0
search is cool

Document 1
search is fun

Document 2
search is fun
for everyone

Lexicon

| | |
|----------|---|
| search | • |
| cool | • |
| fun | • |
| everyone | • |

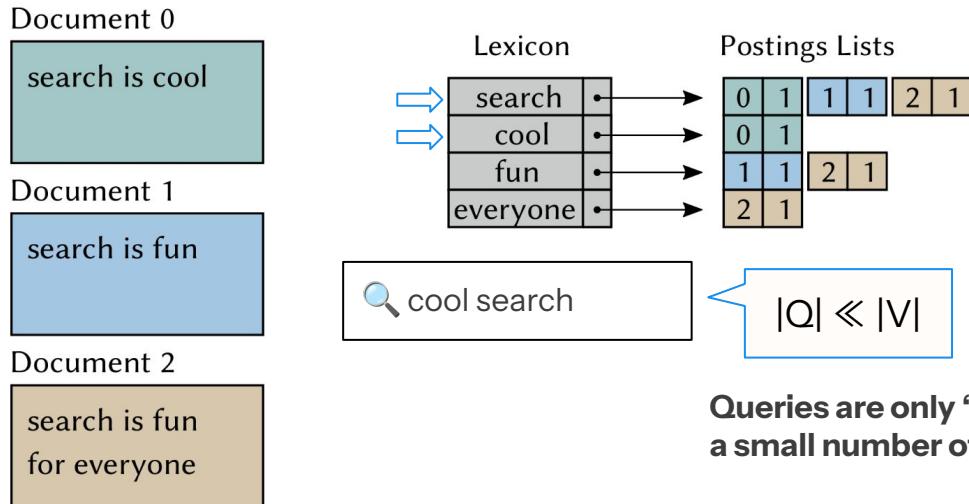
Postings Lists

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 2 | 1 |
| 0 | 1 | | | | |
| 1 | 1 | | | 2 | 1 |
| 2 | 1 | | | | |

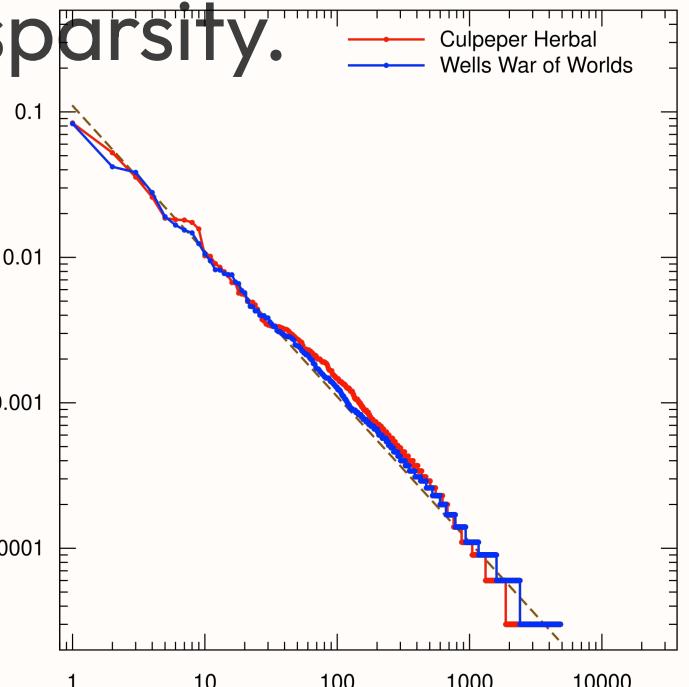
$$|P_{\text{search}}| \ll |C|$$

Terms are only related to a small number of documents
Well... Mostly (see stopwords)

The key characteristic that makes inverted indexes work is sparsity.



Up to now, we've relied on the presence of terms in documents/queries to enforce sparsity.



This works because of **Zipf's Law** – very few terms typically have high frequency in a corpus.

A plot of the frequency of each word as a function of its frequency rank for two English language texts. CC BY-SA 4.0

Working Example

Query: F1 winner

Document: Max Verstappen says 3rd Formula One world championship title is his 'best one' so far

LOSAIL, Qatar (AP) — Max Verstappen believes his third Formula One title is his best yet.

Clinching the championship in a sprint race Saturday in Qatar didn't pack the emotional impact of his dramatic, controversial last-lap overtake of Lewis Hamilton for the 2021 title. Still, the Red Bull driver thinks his relentlessly consistent 2023 season has been his greatest so far.

"This one is the best one," Verstappen said. "I think the first one was the most emotional one because that's when your dreams are fulfilled in Formula One. But this one definitely in my opinion has been my best year also for consecutive wins and stuff. The car itself has been probably in the best shape as well. This one is probably (the one) I'm most proud of in a way because of consistency."

...

Source: AP Online News

Term-Based Representations

Maps a query and document to a sparse “bag-of-words” representation using:

- **TF** (importance of term to the document, based on repetition)
- **IDF** (relative importance of term to the query, based on how many documents it appears in)
- **Other Lexical Signals** (e.g., document length for normalization, etc.)

$$\sum_{t \in q} \log \left(\frac{N}{df_t} \right) \cdot \frac{(k_1 + 1) \cdot tf_{td}}{k_1 \cdot \left(1 - b + b \cdot \left(\frac{L_d}{L_{avg}} \right) \right) + tf_{td}}$$

Term-Based Representations

Maps a query and document to a sparse “bag-of-words” representation using:

- **TF** (importance of term to the document, based on repetition)
- **IDF** (relative importance of term to the query, based on how many documents it appears in)
- **Other Lexical Signals** (e.g., document length for normalization, etc.)

Query: { f1: 10, win: 3 }

Document: { verstappen: 17, race: 13, one: 11, titl: 10,
formula: 8, grand: 7, prix: 7, saturday: 6,
qatar: 6, win: 5, sunday: 4, ..., max: 2, ... }

Term-Based Representations

Problems with term-based representations:

(1) TF and IDF aren't always good estimators of term importance

Example: “Saturday” probably isn’t more important than “Sunday” in the article
(if anything, “Sunday” is more important to the document – that’s when the race was)

Term-Based Representations

Problems with term-based representations:

(1) TF and IDF aren't always good estimators of term importance

Example: “Saturday” probably isn’t more important than “Sunday” in the article
(if anything, “Sunday” is more important to the document – that’s when the race was)

(2) Lexical mismatch: variations of terms that appear in the query/document

Example: “F1” is very important in the article, but isn’t mentioned
(similar terms are, though, e.g., “Formula One”)

**Learned Sparse Retrieval (LSR) uses
neural networks to produce better
bag-of-words representations**

(addressing the aforementioned issues)

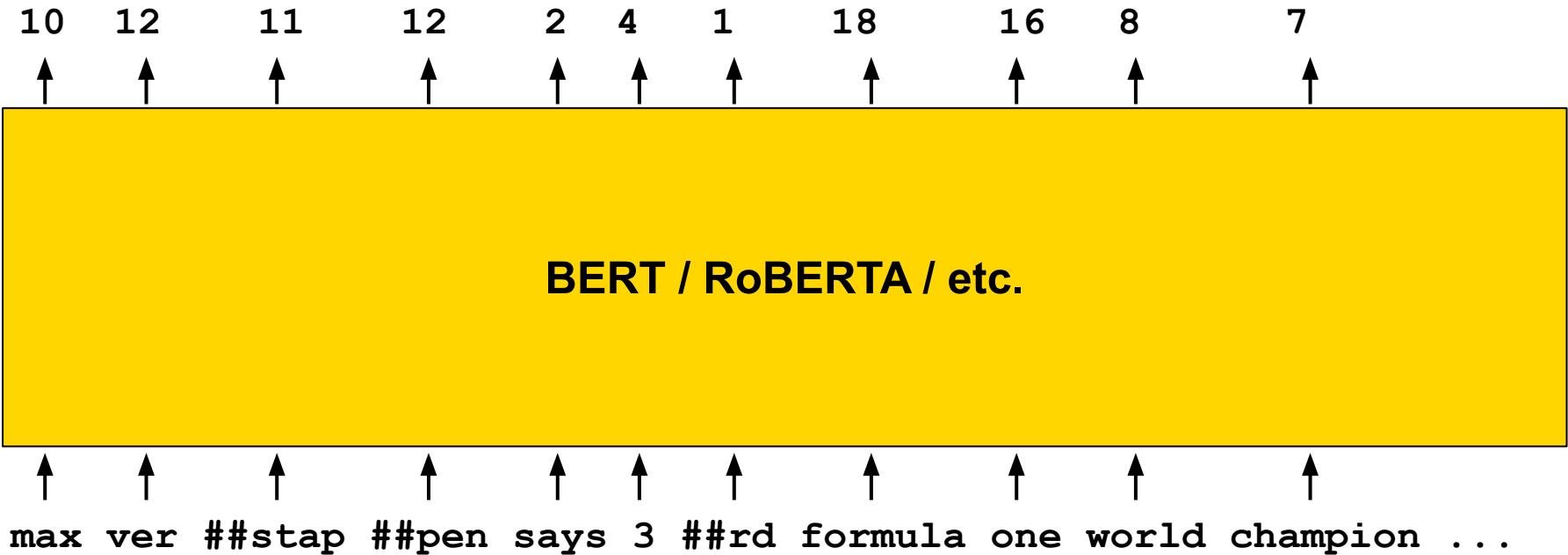
Learned Sparse Retrieval

Three key techniques:

- (1) Term weighting** (addresses importance estimation)
- (2) Expansion** (addresses lexical mismatch)
- (3) Sparsification** (manages compute & storage efficiency of (2))

(1) Term Weighting

Main idea: Use a neural network to estimate the “importance” of each token.
Implemented as a token-level prediction task.



(1) Term Weighting

Main idea: Use a neural network to estimate the “importance” of each token.

```
{formula: 18, one: 16, ver: 12, ##pen: 12, ##stap: 11, ...}
```

10 12 11 12 2 4 1 18 16 8 7



BERT / RoBERTA / etc.

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
max ver ##stap ##pen says 3 ##rd formula one world champion ...

(1) Term Weighting

Term weighting can be applied to the document [1] and/or the query [2].
They can even learn what to remove (weight=0) [3]

[1] Zhuyun Dai, Jamie Callan. **Context-Aware Sentence/Passage Term Importance Estimation For First Stage Retrieval.** arxiv 2019. [link](#)

[2] Sean MacAvaney, Franco Maria Nardini, Raffaele Perego, Nicola Tonellotto, Nazli Goharian, Ophir Frieder. **Expansion via Prediction of Importance with Contextualization.** SIGIR 2020. [Link](#)

[3] Mackenzie et al. **Efficiency Implications of Term Weighting for Passage Retrieval.** SIGIR 2020. [link](#)

(2) Expansion

Goal: Identify new terms to add to the document (and estimate their importance)

Two main approaches:

(2a) External Expansion

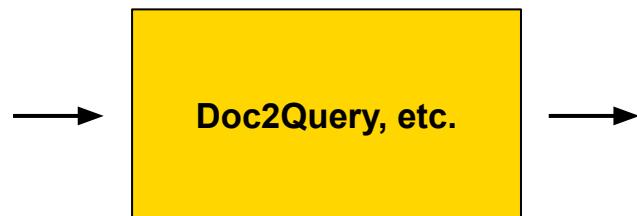
(2b) Masked Language Modeling (MLM) Expansion

(2) Expansion

(2a) External Expansion

Use another model (such as Doc2Query [1]) to add expansion tokens to document [2], then apply term weighting.

“Max Verstappen says
3rd Formula One
world championship
title is his 'best one' so
far...”



- How many F1 championships has
Verstappen won?

- What did Max Verstappen say
about winning his third racing
championship?

...

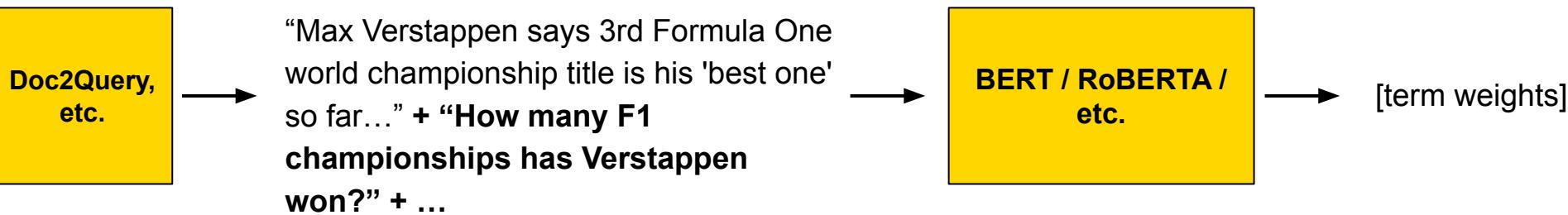
[1] Rodrigo Nogueira, Wei Yang, Jimmy Lin, Kyunghyun Cho. **Document Expansion by Query Prediction**. arxiv 2019. [link](#)

[2] Antonio Mallia, Omar Khattab, Nicola Tonellotto, Torsten Suel. **Learning Passage Impacts for Inverted Indexes**. SIGIR 2021. [link](#)

(2) Expansion

(2a) External Expansion

Use another model (such as Doc2Query [1]) to add expansion tokens to document [2], then apply term weighting.



[1] Rodrigo Nogueira, Wei Yang, Jimmy Lin, Kyunghyun Cho. **Document Expansion by Query Prediction**. arxiv 2019. [Link](#)

[2] Antonio Mallia, Omar Khattab, Nicola Tonellotto, Torsten Suel. **Learning Passage Impacts for Inverted Indexes**. SIGIR 2021. [link](#)

(2) Expansion

(2b) Masked Language Modeling Expansion

Use the model's Masked Language Modelling head to get expansion terms.

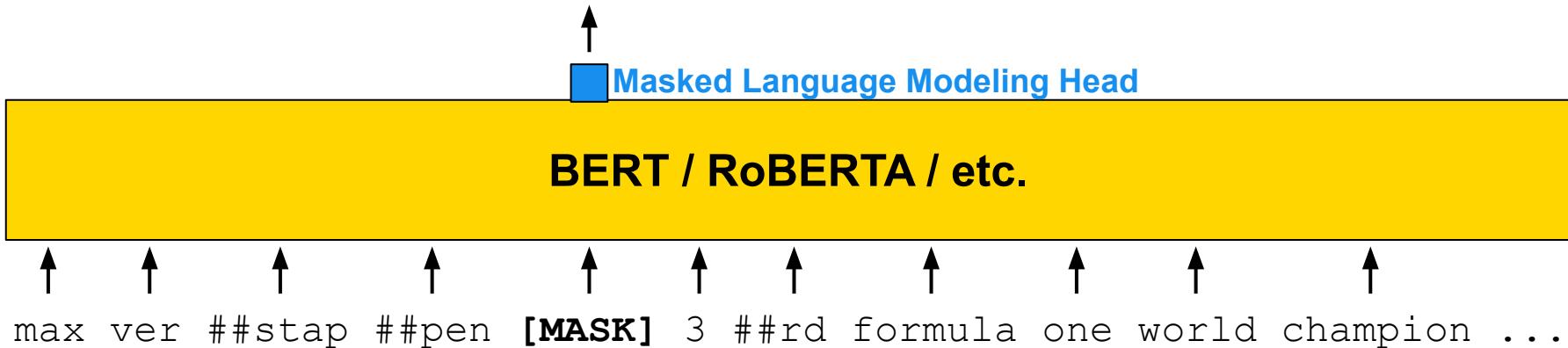
(2) Expansion

(2b) Masked Language Modeling Expansion

Use the model's Masked Language Modelling head to get expansion terms.

Review: Pre-training a BERT-like language model

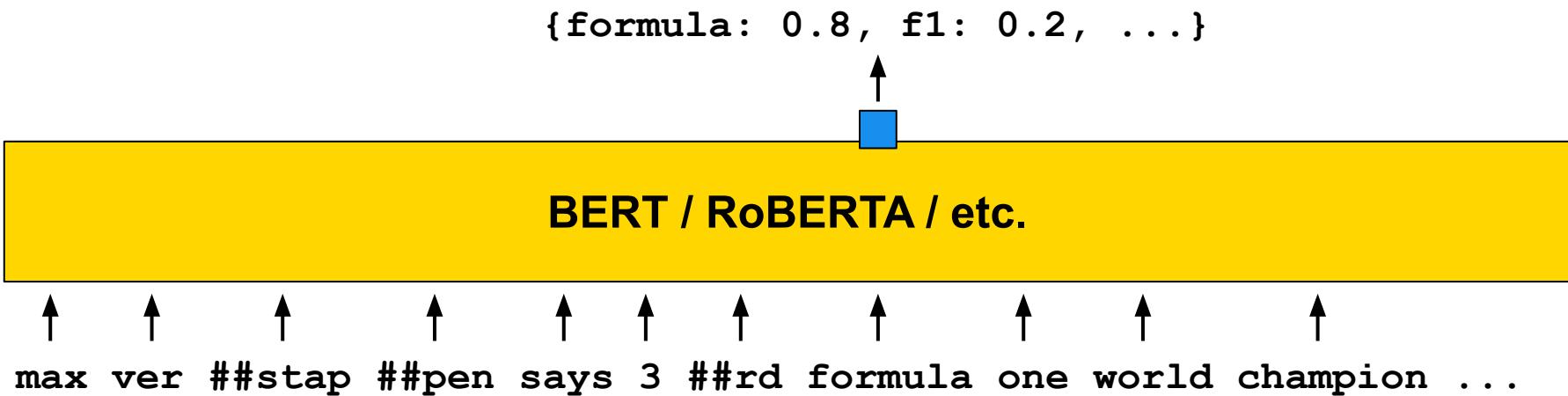
```
{says: 0.3, suggests: 0.2, said: 0.2, tweeted: 0.1, ...}
```



(2) Expansion

(2b) Masked Language Modeling Expansion

Use the model's Masked Language Modelling head to get expansion terms.

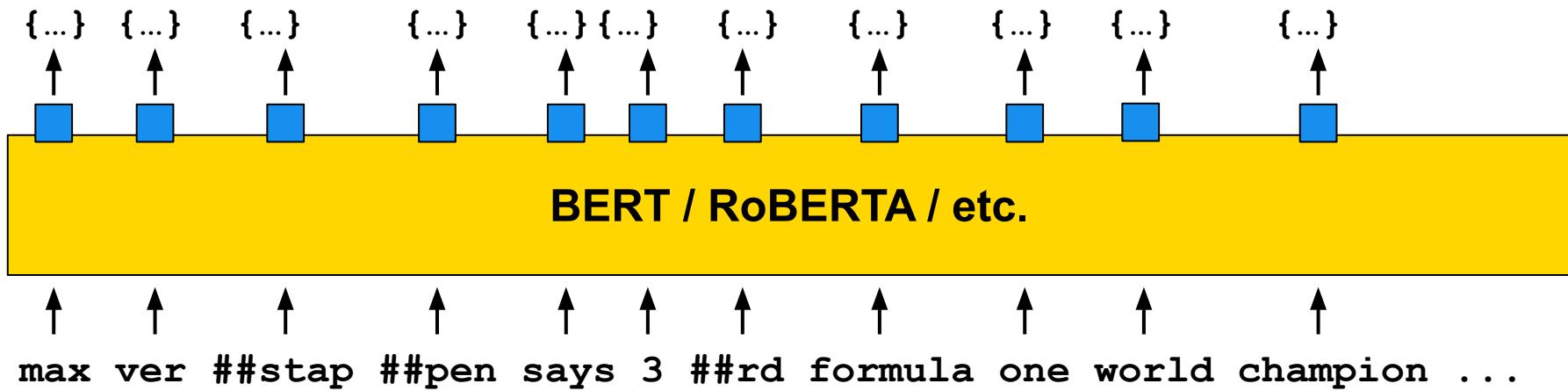


(2) Expansion

(2b) Masked Language Modeling Expansion

Use the model's Masked Language Modelling head to get expansion terms.

```
{formula: 18, one: 16, ver: 12, ##pen: 12, f1: 10, ...}
```



(2) Expansion

(2b) MLM Expansion can be learned end-to end, and can be applied to the document [1] and/or the query [2].

[1] Sean MacAvaney, Franco Maria Nardini, Raffaele Perego, Nicola Tonellotto, Nazli Goharian, Ophir Frieder. **Expansion via Prediction of Importance with Contextualization.** SIGIR 2020. [link](#)

[2] Thibault Formal, Benjamin Piwowarski and Stéphane Clinchant. **SPLADE: Sparse Lexical and Expansion Model for First Stage Ranking.** SIGIR 2021. [Link](#)

[3] Zhuang and Zuccon. **Fast Passage Re-ranking with Contextualized Exact Term Matching and Efficient Passage Expansion.** [link](#)

(3) Sparsification

MLM Expansion actually makes the output vectors dense – it computes an importance score for every token in the lexicon (10's of thousands of dimensions).

We could index these vectors in dense vector stores, but they're very large, and this makes retrieval slow.

Instead, we sparsify, allowing vectors to be used in typical inverted indexes (e.g., Lucene)

Sparsification approaches:

(3a): Top-K Pruning [1] (post-hoc pruning of lowest vector dimensions)

(3b): FLOPS Regularisation [2] (end-to-end optimisation to push dimensions to zero)

(3c): DF-FLOPS Regularisation [3] (objective to reduce number of terms per document)

[1] Sean MacAvaney, Franco Maria Nardini, Raffaele Perego, Nicola Tonellotto, Nazli Goharian, Ophir Frieder. **Expansion via Prediction of Importance with Contextualization**. SIGIR 2020. [link](#)

[2] Thibault Formal, Benjamin Piwowarski and Stéphane Clinchant. **SPLADE: Sparse Lexical and Expansion Model for First Stage Ranking**. SIGIR 2021. [Link](#)

[3] Porco et al. **An Alternative to FLOPS Regularization to Effectively Productionize SPLADE-doc**. SIGIR 2025. [link](#)

Which LSR methods produce better relevance estimates?

In general:

Query Weighting (query expansion only helps a little when document is expanded)

+

Document MLM Expansion

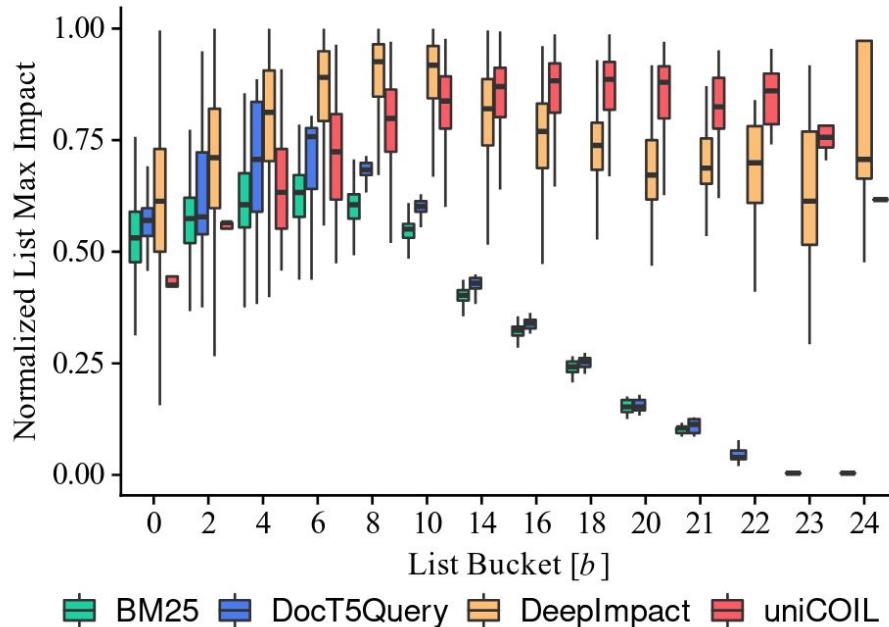
+

Regularisation (for optimal trade-offs) or Top-K (for flexibility)

(Find out more in the **Neural Lexical Search with Learned Sparse Retrieval** tutorial this afternoon.)

How do LSR methods affect retrieval algorithms?

Term Weighting greatly affects impact distributions



Normalized maximum list impact distribution stratified by list length buckets
 $b \in [2^b, 2^{b+1})$.

Wacky Weights in Learned Sparse Representations and the Revenge of Score-at-a-Time Query Evaluation

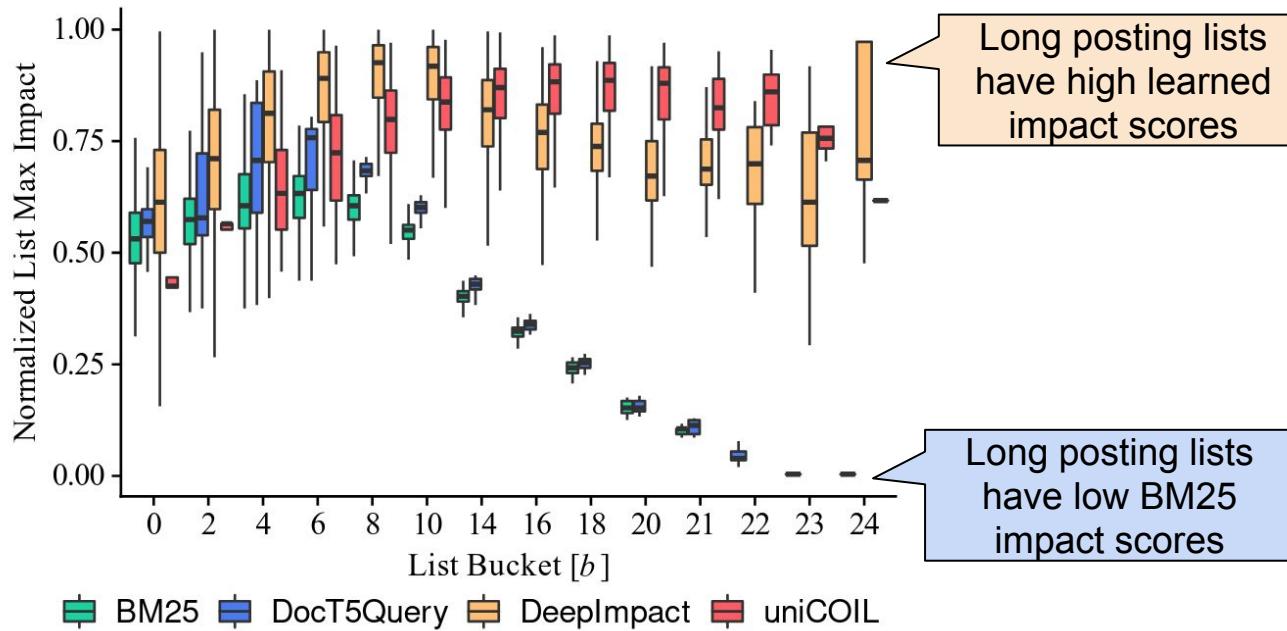
Joel Mackenzie,¹ Andrew Trotman,² Jimmy Lin³

¹ School of Computing and Information Systems, The University of Melbourne, Australia

² Department of Computer Science, University of Otago, Dunedin, New Zealand

³ David R. Cheriton School of Computer Science, University of Waterloo, Ontario, Canada

Term Weighting greatly affects impact distributions



**Wacky Weights in Learned Sparse Representations and the
Revenge of Score-at-a-Time Query Evaluation**

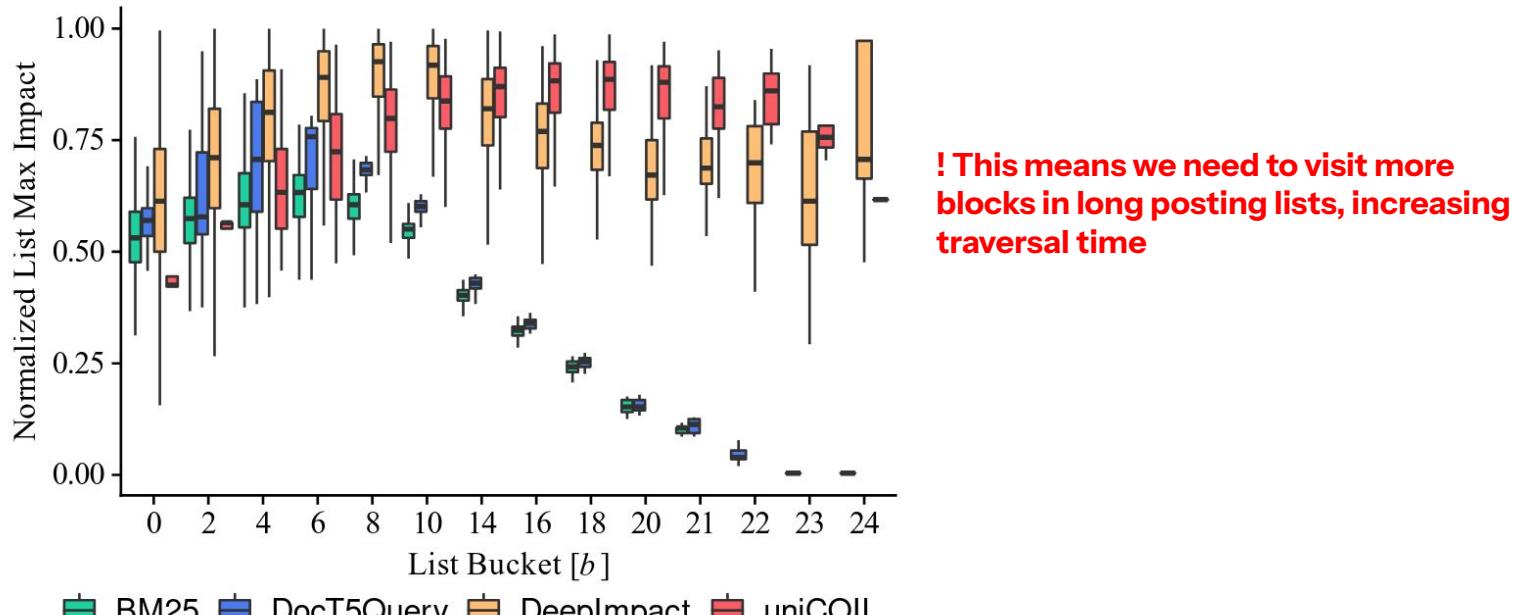
Joel Mackenzie,¹ Andrew Trotman,² Jimmy Lin³

¹ School of Computing and Information Systems, The University of Melbourne, Australia

² Department of Computer Science, University of Otago, Dunedin, New Zealand

³ David R. Cheriton School of Computer Science, University of Waterloo, Ontario, Canada

Term Weighting greatly affects impact distributions



**Wacky Weights in Learned Sparse Representations and the
Revenge of Score-at-a-Time Query Evaluation**

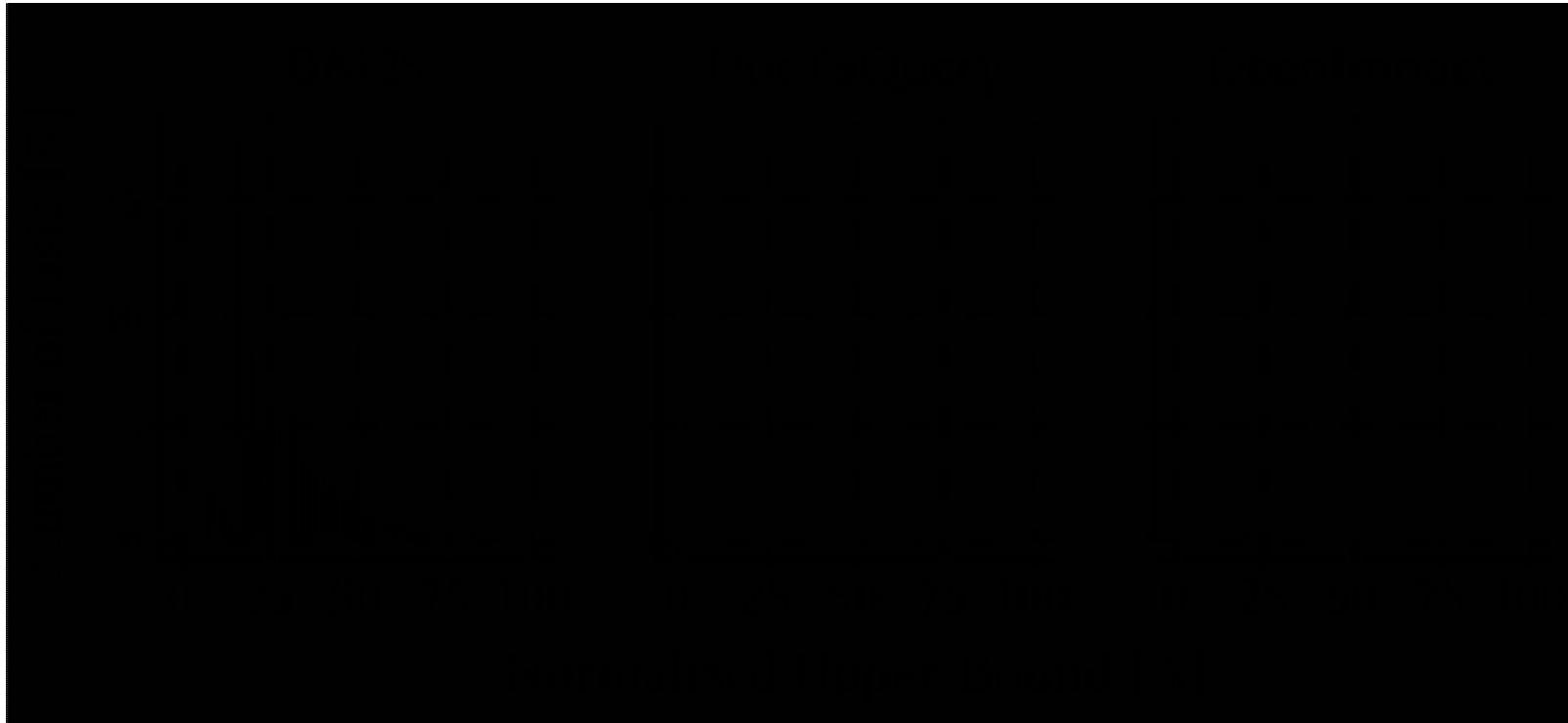
Joel Mackenzie,¹ Andrew Trotman,² Jimmy Lin³

¹ School of Computing and Information Systems, The University of Melbourne, Australia

² Department of Computer Science, University of Otago, Dunedin, New Zealand

³ David R. Cheriton School of Computer Science, University of Waterloo, Ontario, Canada

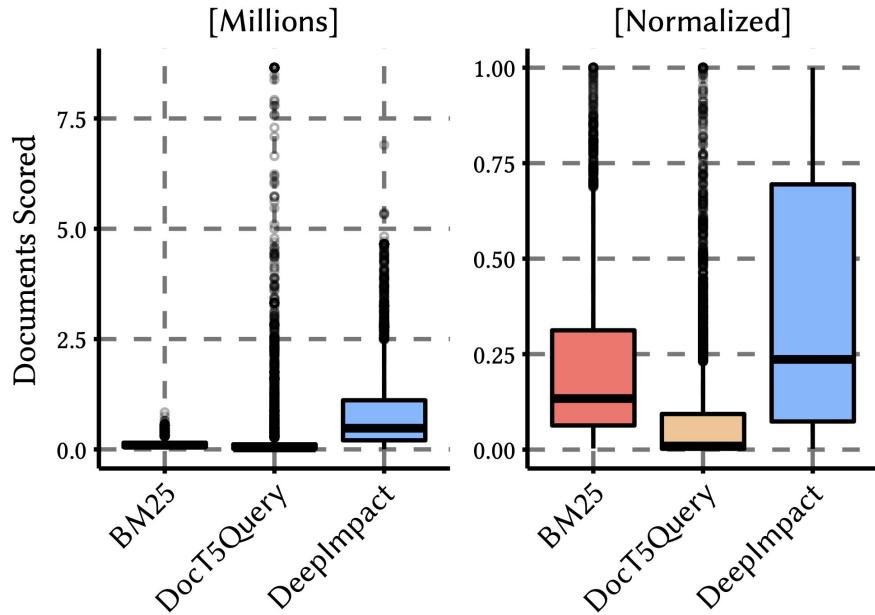
Term Weighting greatly affects impact distributions



Distributions Make or Break Efficiency

Efficient top-k processing algorithms use term upper-bound scores to bypass documents which cannot score highly.

- These algorithms find it more difficult to effectively prune the search space with the impact distributions from learned sparse models.
- More documents are scored; query processing is slower.



There are very long posting lists, too.

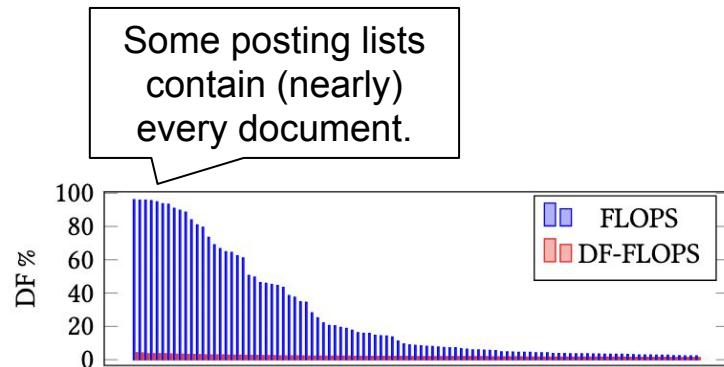


Figure 2: Top 100 DF% in representations produced by SPLADE-Doc with FLOPS and DF-FLOPS on a sample of 100K passages.

There are very long posting lists, too. Why?

Some tokens are repurposed and included in many queries and documents.

Ends up being partially a pseudo “dense” vector.

androgen receptor define →

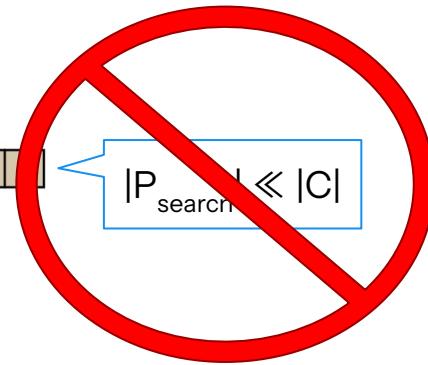
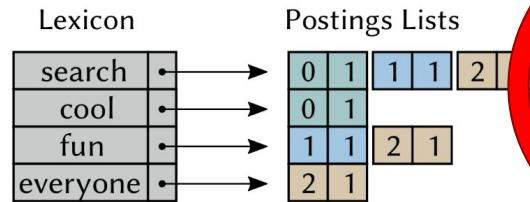
```
('##rogen', 251) ('receptor', 242) ('and', 225) ('receptors', 189)
('hormone', 179) ('definition', 162) ('meaning', 99) ('genus', 89)
('is', 70) (',', 68) ('define', 59) ('the', 56) ('drug', 53) ('for', 46)
('ring', 38) ('gene', 37) ('are', 32) ('god', 25) ('what', 18) ('##rus', 15)
('purpose', 12) ('defined', 10) ('doing', 8) ('a', 4) ('goal', 4)
```

There are very long posting lists, too.

Document 0
search is cool

Document 1
search is fun

Document 2
search is fun
for everyone



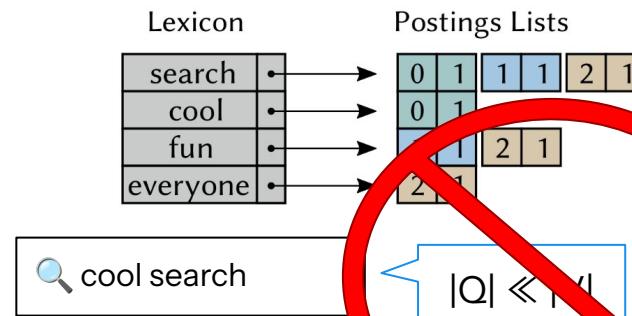
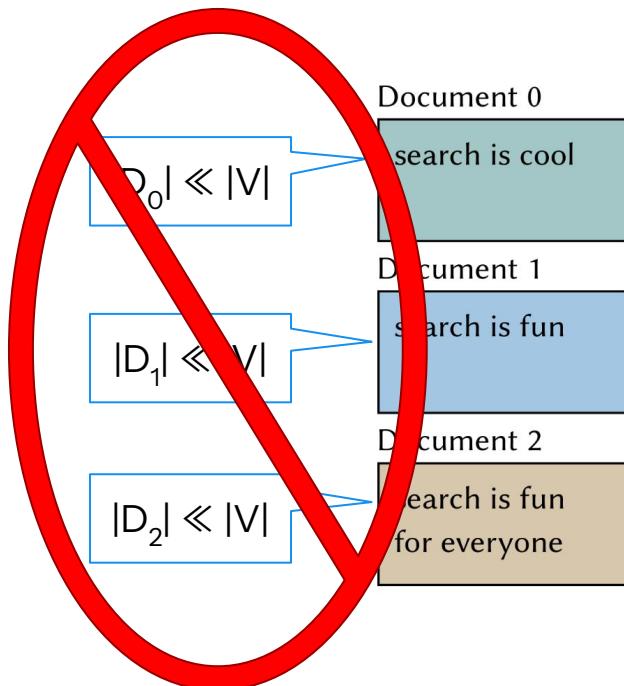
Assumption no longer holds.

Expansion causes much less sparsity in queries/docs

MSMARCO Passages (dev queries)

| Method | V | Terms in Documents | | Terms in Queries | |
|---------------|---------|--------------------|--------|------------------|--------|
| | | Total | Unique | Total | Unique |
| BM25 | 2660824 | 39.8 | 30.1 | 5.9 | 5.8 |
| BM25-T5 | 3929111 | 224.7 | 51.1 | 5.9 | 5.8 |
| DeepImpact | 3514102 | 4010.0 | 71.1 | 4.2 | 4.2 |
| uniCOIL-T5 | 27678 | 5032.3 | 66.4 | 686.3 | 6.6 |
| uniCOIL-TILDE | 27646 | 8260.8 | 107.6 | 661.1 | 6.5 |
| SPLADEv2 | 28131 | 10794.8 | 229.4 | 2037.8 | 25.0 |

Expansion causes much less sparsity in queries/docs



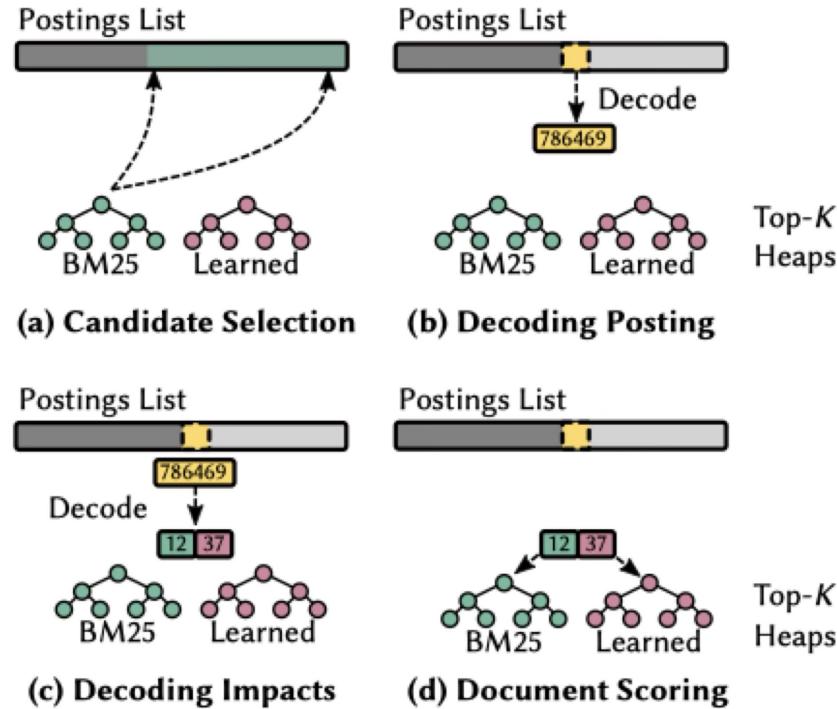
Assumptions no longer hold.

So what can we do about these problems?

Guided Traversal (GT)

It proposes guided traversal to accelerate top-k processing with learned sparse models.

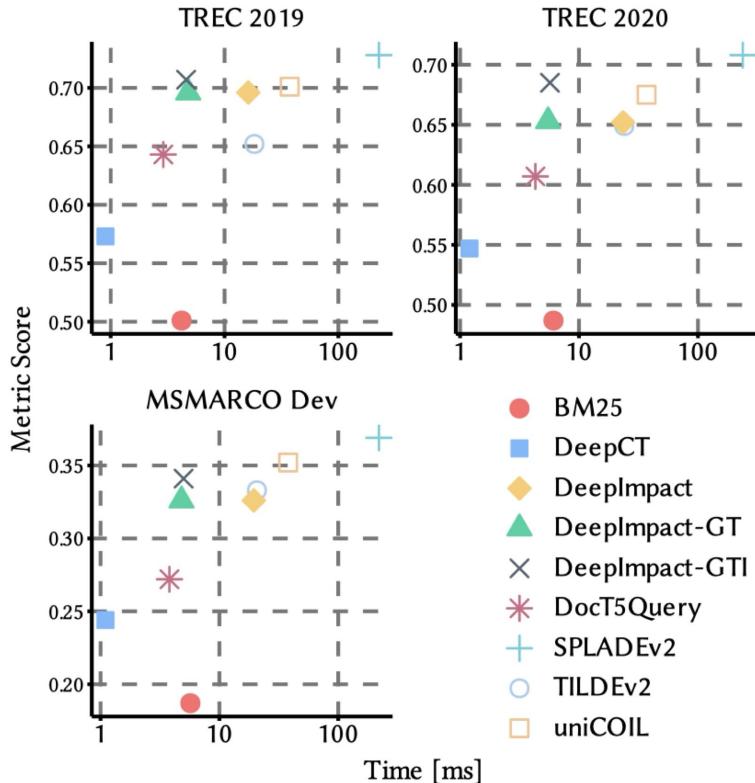
- The original BM25 score for each document (over a DocT5Query expanded index) is stored alongside the learned (DeeplImpact) score.
- At query time, BM25 is used to guide the index traversal, but scores are computed via the learned model (Guided Traversal — GT).
- The BM25 score can also be interpolated with the DeeplImpact score on-the-fly (Guided Traversal with Interpolation — GTI).



Guided Traversal (GT)

| Strategy | MaxScore | | | |
|---|----------|--------|----------|-------|
| | Mean | Median | P_{99} | RR@10 |
| DocT5Query | 3.3 | 2.8 | 12.2 | 0.272 |
| DeepImpact | 19.5 | 14.0 | 79.6 | 0.326 |
| Dual-Heap | 4.6 | 3.9 | 15.8 | 0.326 |
| Dual-Heap w. Interp. ($\alpha = 0.5$) | 4.6 | 3.9 | 15.8 | 0.335 |

Guided Traversal (GT)



Mallia et al. "Faster learned sparse retrieval with guided traversal." SIGIR 2022.

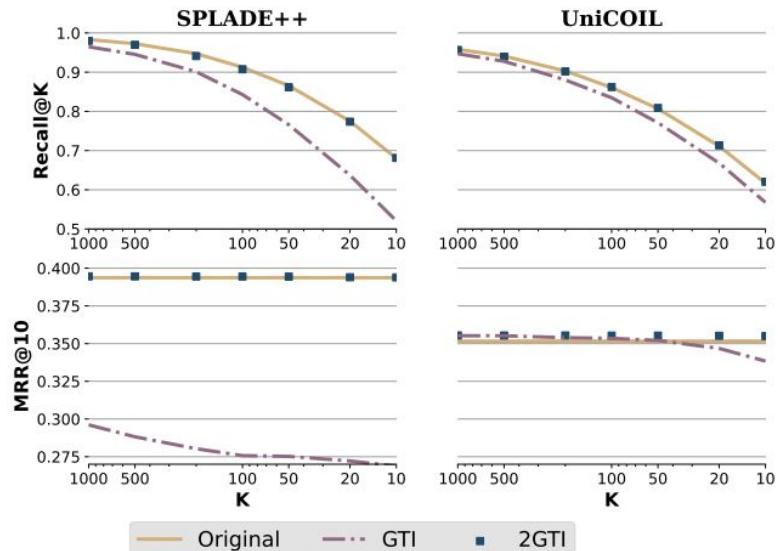
Two-Level Guided Traversal (2GT)

Global pruning – hybrid BM25 + learned upper bounds filter whole posting-list regions.

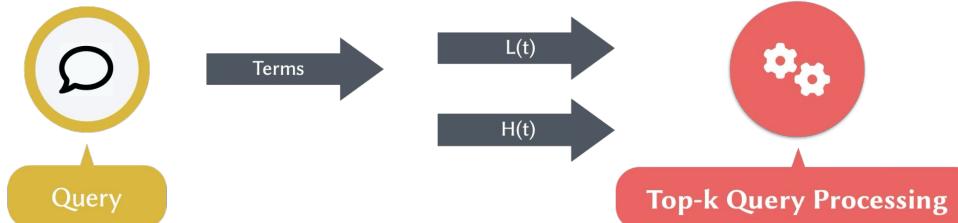
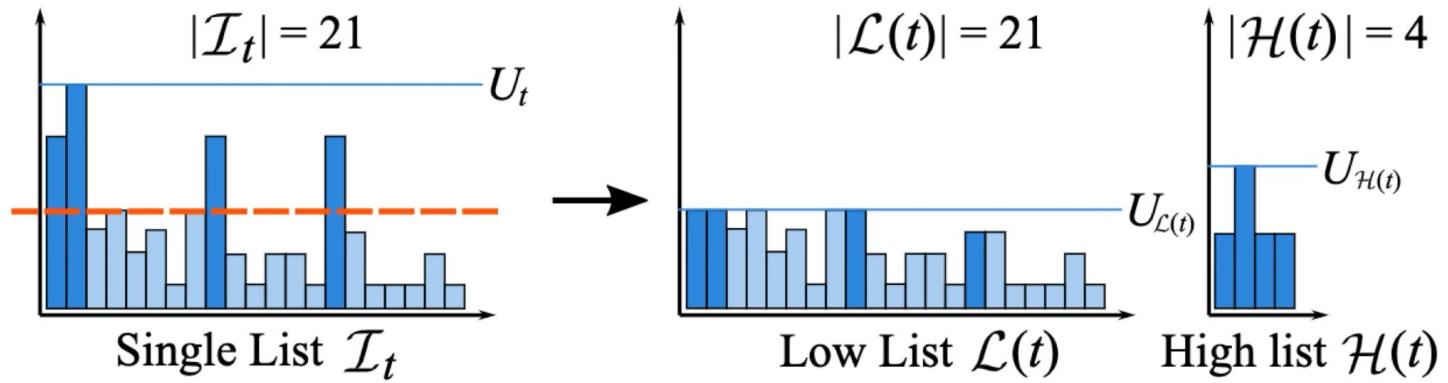
Local pruning – hybrid bounds tighten inside each candidate document.

Alignment smoothing makes BM25 weights denser to match the learned index.

Two tunable coefficients (α for global, β for local) cap BM25's influence and prevent over-aggressive skips.



Postings Clipping



Priming can be applied whenever any high-impact list contains k or more postings

$$\theta_0 = \max\{U_{\mathcal{L}(t)} \mid t \in Q \wedge |\mathcal{H}(t)| \geq k\}$$

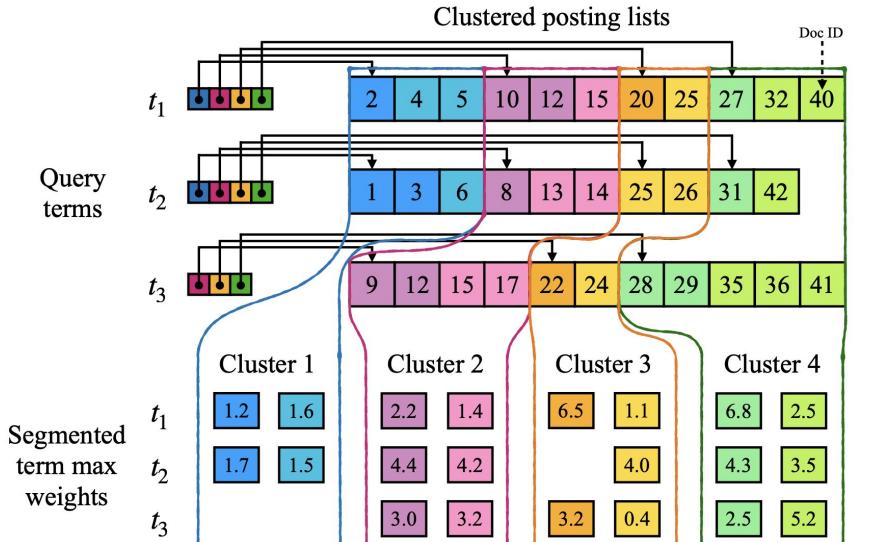
can be used as a priming value for the heap bound, without risking the integrity of the top-k answers.

ASC

Segmented bounds: slice each cluster into n random segments, store per-segment max weights \rightarrow tighter MaxSBound / AvgSBound

Two-level test: prune cluster if $\text{MaxSBound} \leq \theta/\mu$ and $\text{AvgSBound} \leq \theta/n$; else dive to doc-level pruning at θ/η .

Parameters: $0 < \mu \leq \eta \leq 1$. Pick μ for aggressiveness, η (often 1) for probabilistic safety.



(a) Cluster skipping index with 2 weight segments per cluster

Seismic

An approximate retrieval solution that trades off exact search for efficiency.

It relies on:

- Concentration of Importance
- Static Document Pruning
- Block Upper Bounds

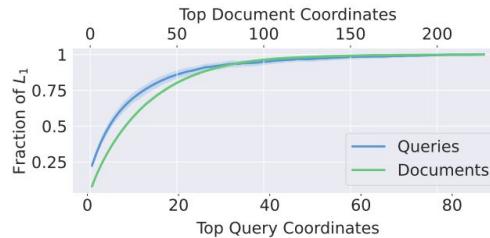


Figure 1: Fraction of L_1 mass preserved by keeping only the top non-zero entries with the largest absolute value.

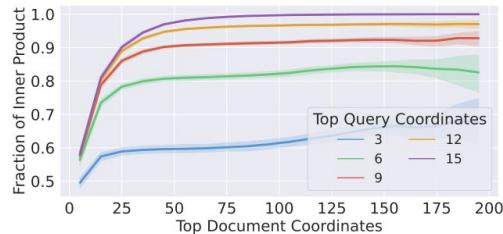
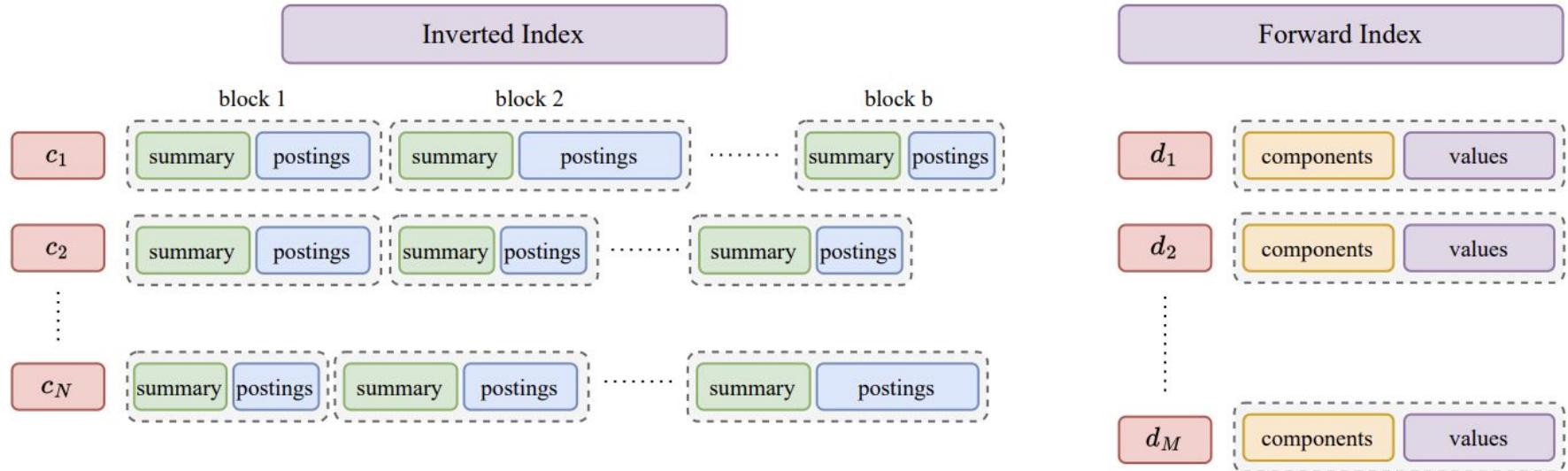
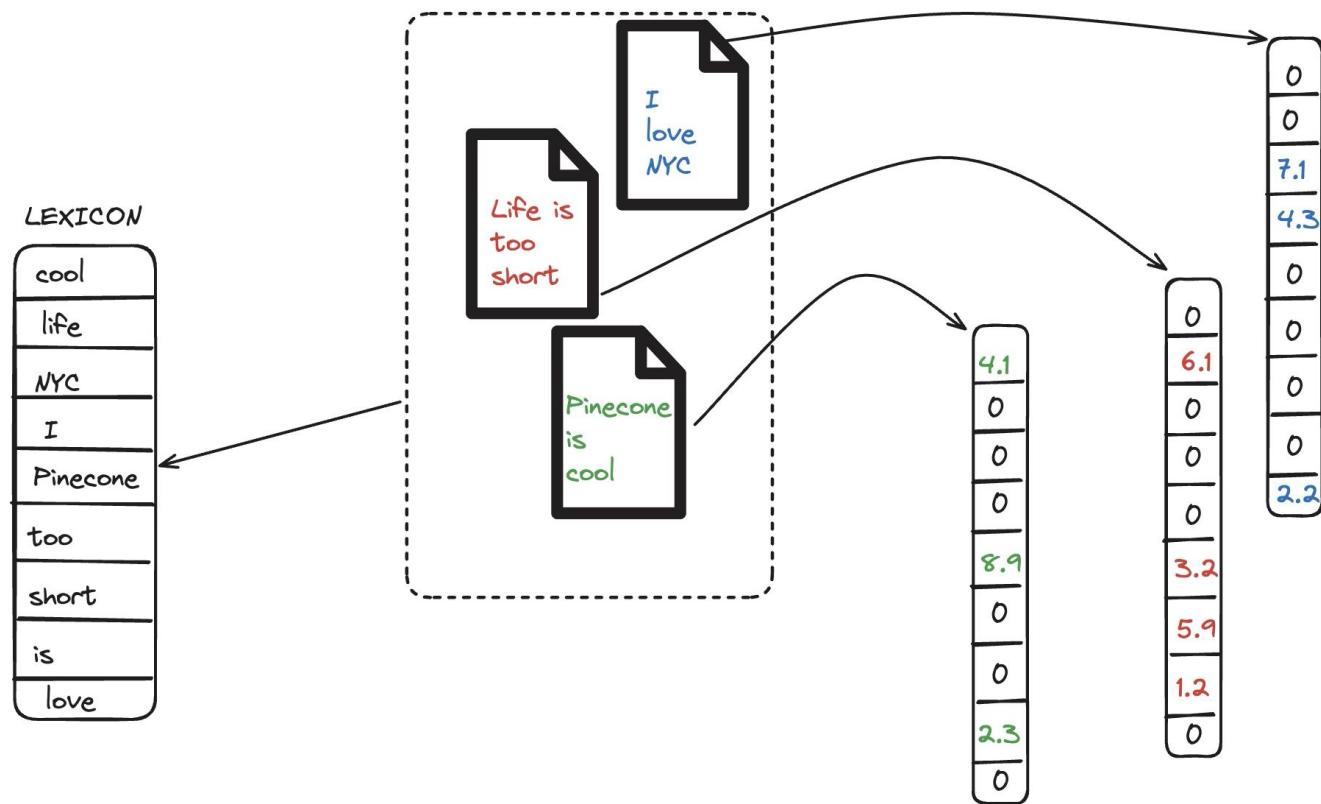


Figure 2: Fraction of inner product (with 95% confidence intervals) preserved by inner product between the top query and document coordinates with the largest absolute value.

Seismic



Documents as Sparse Vectors



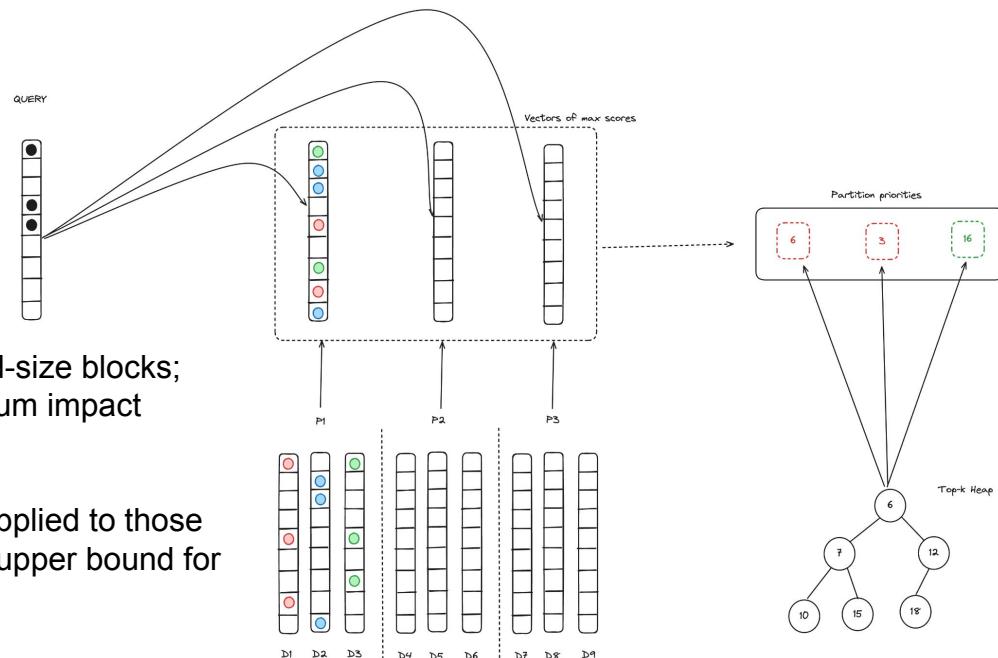
Block-Max Pruning

Block partitioning: The document ID space is split into fixed-size blocks; each block stores a pre-computed vector of its terms' maximum impact scores.

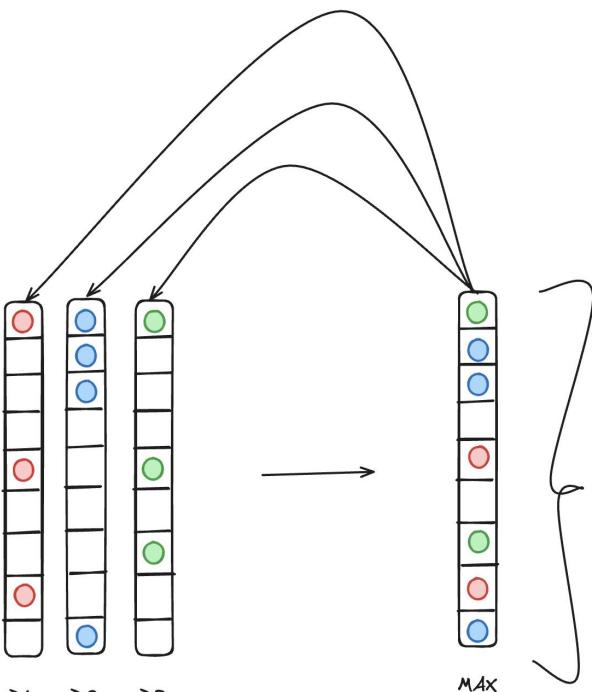
Per-block upper bounds: At query time, term weights are applied to those block-max vectors and summed, producing an overall score upper bound for every block.

Priority-driven evaluation: Blocks are visited in descending order of their upper-bound scores (lazy sorting) until a stopping criterion is satisfied.

Hybrid access: When a block is chosen, lookup switches to a forward-style structure embedded alongside the inverted lists, enabling fast in-block scoring without full postings scans.



Clustering of Documents

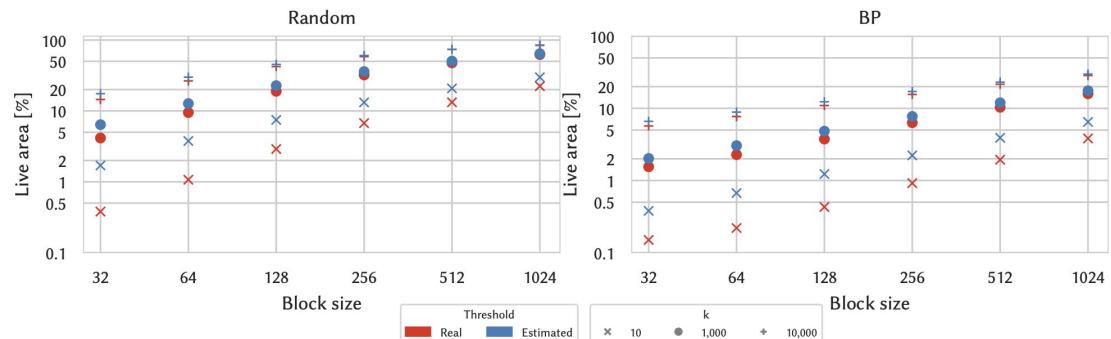


Imagine assigning consecutive docIDs to similar documents.

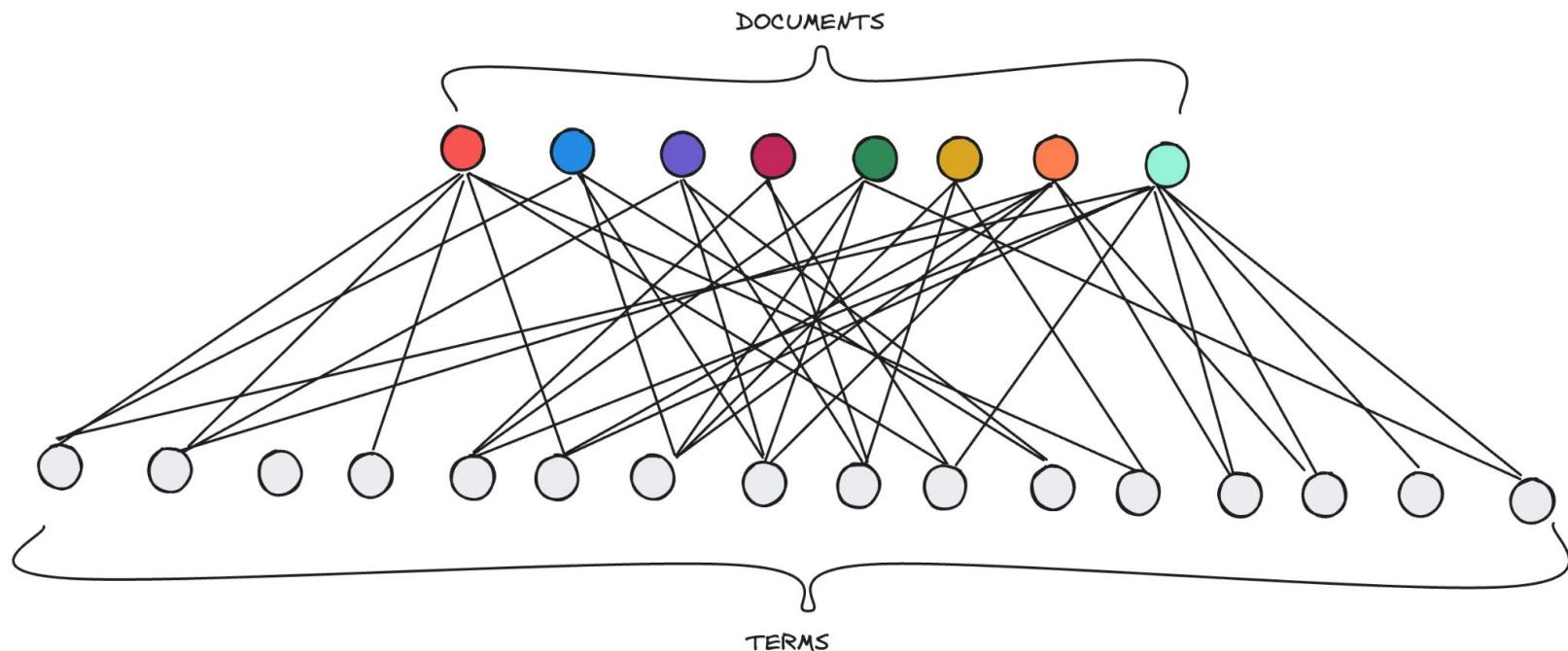
First focus on better compression.

Recently, more focus on faster query processing.

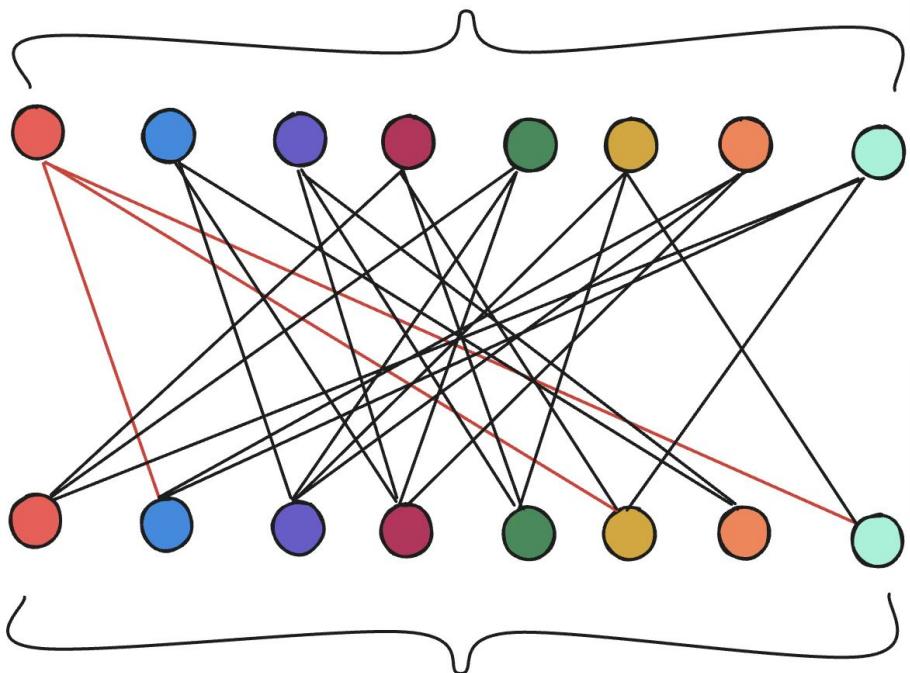
MAXIMIZE SPARSITY



Bipartite Graph Partitioning



Bipartite Graph Partitioning



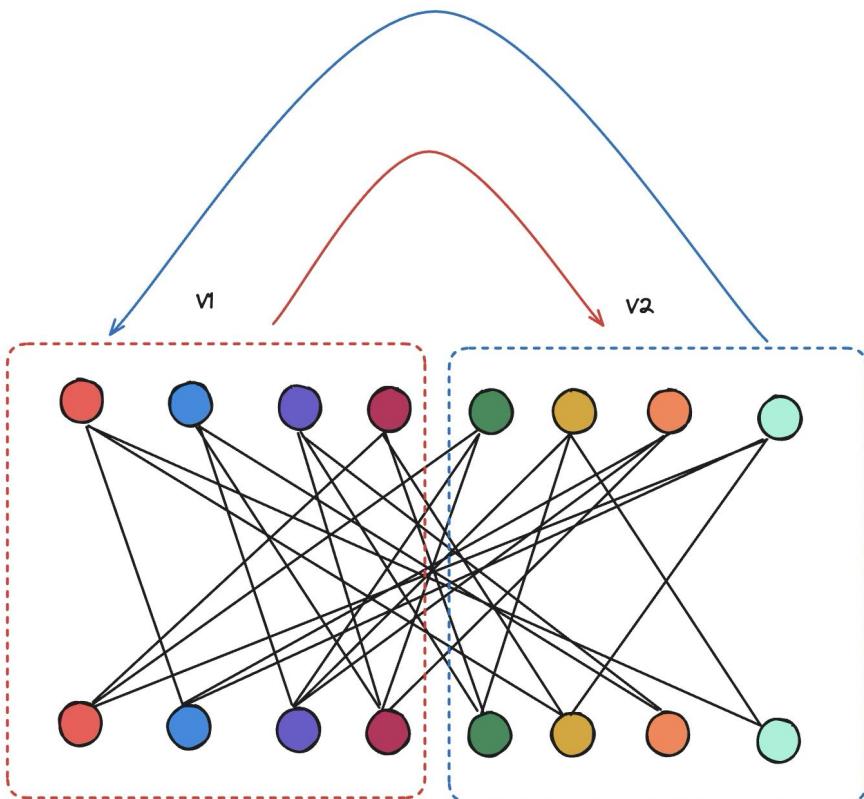
$\deg(x)$ is the degree of a node

n is the number of distinct neighbors

The average log gap cost can be proxied with

$$\deg(x) \times \log\left(\frac{n}{\deg(x) + 1}\right)$$

Graph Bisection



Bisect the graph in two sets

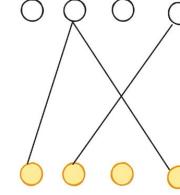
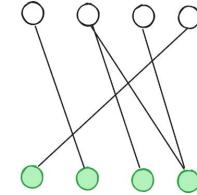
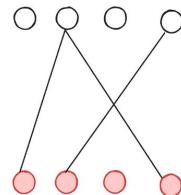
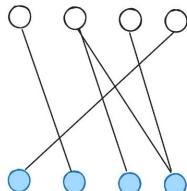
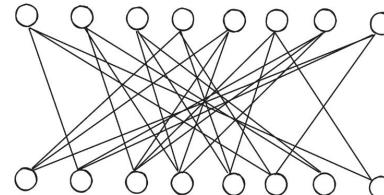
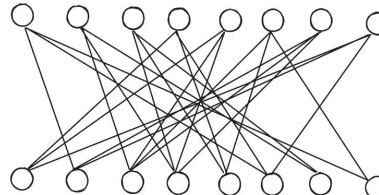
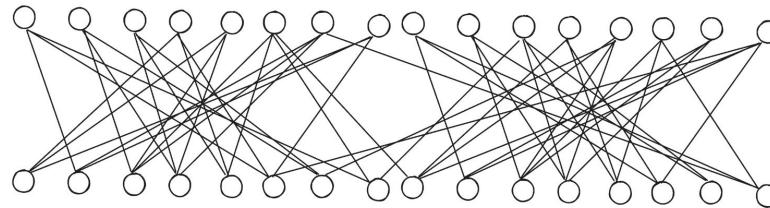
Compute move gains of the vertices. The difference in average logarithmic gap length between remaining and moving

$$\left[deg_1(x) \times \log\left(\frac{n_1}{deg_1(x)+1}\right) + deg_2(x) \times \log\left(\frac{n_2}{deg_2(x)+1}\right) \right] - \left[(deg_1(x)-1) \times \log\left(\frac{n_1}{deg_1(x)}\right) + (deg_2(x)+1) \times \log\left(\frac{n_2}{deg_2(x)+2}\right) \right]$$

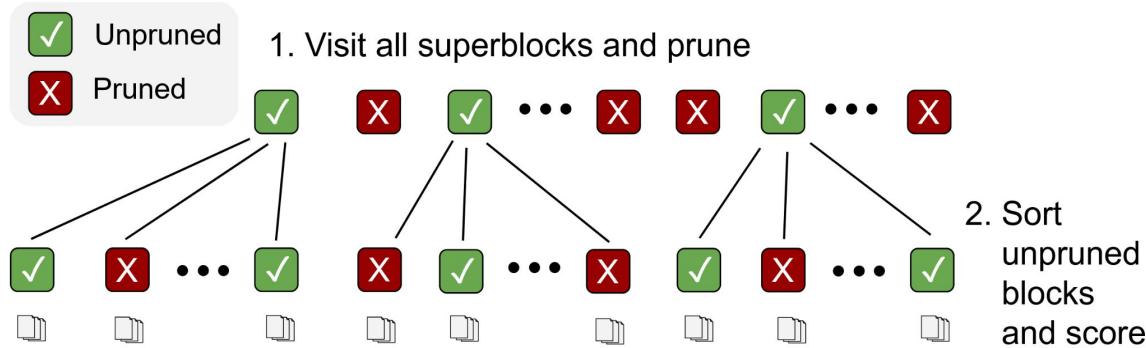
Swap vertices between the two sets

Repeat recursively until stopping condition triggers

Recursive Graph Bisection



Dynamic Superblock Pruning



Two-tier index: group consecutive document blocks into fixed-size superblocks (e.g., 64 blocks)

Top-down test: first bound each superblock; prune it if both max- and avg-score $\leq \theta/\mu, \theta/\eta$, then descend to surviving blocks

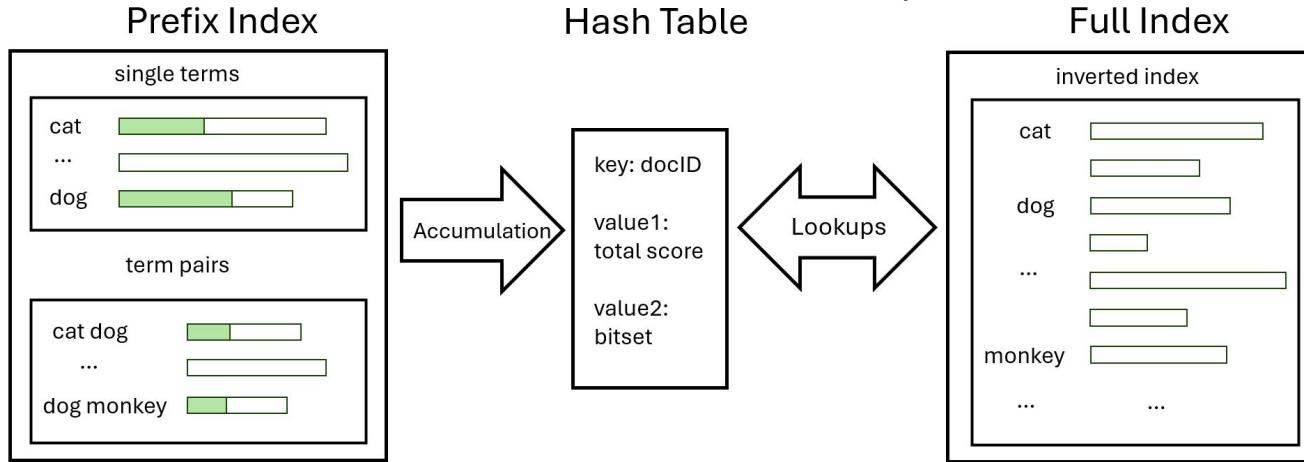
Dynamic Superblock Pruning

Table 1: Mean response time (ms) and mean reciprocal rank (MRR@10) at a fixed Recall@ k budget for SPLADE

| Recall Budget | 99% | | 99.5% | | 99.9% | | Rank-Safe | |
|------------------|--------------|------|--------------|------|--------------|------|-------------|------|
| | MRT | MRR | MRT | MRR | MRT | MRR | MRT | MRR |
| $k=10$ | | | | | | | | |
| MaxScore | – | – | – | – | – | – | 75.7 (35x) | 38.1 |
| ASC | 4.70 (7.5x) | 37.9 | 5.59 (7.8x) | 38.1 | 6.44 (8.2x) | 38.1 | 7.19 (3.3x) | 38.1 |
| Seismic | 2.06 (3.3x) | 38.1 | 2.57 (3.6x) | 38.2 | 3.01 (3.8x) | 38.4 | – | – |
| BMP | 1.44 (2.3x) | 38.1 | 1.49 (2.1x) | 38.1 | 1.88 (2.4x) | 38.2 | 2.70 (1.3x) | 38.1 |
| SP | 0.629 | 37.7 | 0.715 | 37.9 | 0.785 | 38.1 | 2.15 | 38.1 |
| $k=1000$ | | | | | | | | |
| MaxScore | – | – | – | – | – | – | 124 (12x) | 38.1 |
| ASC | 15.8 (9.1x) | 38.1 | 18.9 (9.4x) | 38.1 | 25.4 (5.5x) | 38.1 | 33.5 (3.2x) | 38.1 |
| Seismic | 5.72 (3.3x) | 38.3 | 7.18 (3.6x) | 38.4 | 10.5 (2.3x) | 38.4 | – | – |
| BMP | 4.99 (2.9x) | 38.2 | 5.25 (2.6x) | 38.2 | 7.26 (1.6x) | 38.2 | 13.9 (1.3x) | 38.1 |
| SP | 1.74 | 37.9 | 2.01 | 37.9 | 4.64 | 38.2 | 10.5 | 38.1 |

SPRAWL

Sorted PRefix Access With Lookups



It uses a two-tier in-memory index:

- A prefix index that stores top-scoring postings (sorted by impact scores in descending order) for frequent terms and pairs of terms.
- A standard inverted index that containing all standard (single-term) postings and that efficiently supports random lookups.

Rough notes/stuff we need

- Relevant links to bibliographies
- Relevant links to PISA/PyTerrier/Slack Channes/Resources/etc
- Guide on contributing to PISA
- Other codebases of interest (broader PISA/Terrier/etc projects)

Session III: Future Directions / Soapboxes

Joel's Soapbox

Long live the inverted index!

Moving towards Rust.

Sean's Soapbox

The PISA Ecosystem

CIFF

Common Index File Format CIFF is an inverted index exchange format as defined as part of the Open-Source IR Replicability Challenge (OSIRRC) initiative.

We built tools to convert:

- a CIFF blob to a PISA canonical: **ciff2pisa**
- a PISA canonical to a CIFF blob: **pisa2ciff**
- a JSONL file to a CIFF blob: **jsonl2ciff**

```
message Header {  
    int32 version = 1;  
    int32 num_postings_lists = 2;  
    int32 num_docs = 3;  
    int32 total_postings_lists = 4;  
    int32 total_docs = 5;  
    int64 total_terms_in_collection = 6;  
    double average_doclength = 7;  
    string description = 8;  
}  
  
message Posting {  
    int32 docid = 1;  
    int32 tf = 2;  
}  
  
message PostingsList {  
    string term = 1;  
    int64 df = 2;  
    int64 cf = 3;  
    repeated Posting postings = 4;  
}  
  
message DocRecord {  
    int32 docid = 1;  
    string collection_docid = 2;  
    int32 doclength = 3;  
}
```

Figure 1: Protobuf definitions of messages in CIFF.

CIFF-Hub



CIFF Hub

Common Index File Format CIFF is an inverted index exchange format as defined as part of the Open-Source IR Replicability Challenge (OSIRRC) initiative.

The Ciff Hub hosts many indexes and queries for a variety of collections and models.

Python Integration

Trades off some flexibility and efficiency for convenience.

Indexing:

```
● ● ●

from pyterrier_pisa import PisaIndex

# index from your own data:
index = PisaIndex('my-index.pisa')
index.index([
    {'docno': '1', 'text': 'Check out the PISA engine!'},
    {'docno': '2', 'text': 'Here is the Python integration'},
    ...
])

# index from a huggingface dataset:
from datasets import load_dataset
index = PisaIndex('arthur-conan-doyle.pisa')
index.index(load_dataset('macavaney/arthur-conan-doyle'))
```

Python Integration

Trades off some flexibility and efficiency for convenience.

Retrieval:

```
● ● ●

from pyterrier_pisa import PisaIndex
index = PisaIndex('msmarco-passage.pisa')
bm25 = index.bm25()
bm25.search('pisa engine')

#      qid      query      docno      score  rank
# 0      1  pisa engine  7378605  18.529522    0
# 1      1  pisa engine   49001  17.285473    1
# 2      1  pisa engine  3322192  17.025841    2
# 3      1  pisa engine  4800437  16.870358    3
# 4      1  pisa engine  2480271  16.774212    4
# ...
```

Python Integration

Trades off some flexibility and efficiency for convenience.

Sharing Indexes:

```
● ● ●

from pyterrier_pisa import PisaIndex

# share an index to huggingface
PisaIndex.to_hf('macavaney/my-index.pisa')

# load an index from huggingface
PisaIndex.from_hf('macavaney/msmarco-passage.pisa')
```

https://huggingface.co/datasets?other=pyterrier-artifact.sparse_index.pisa

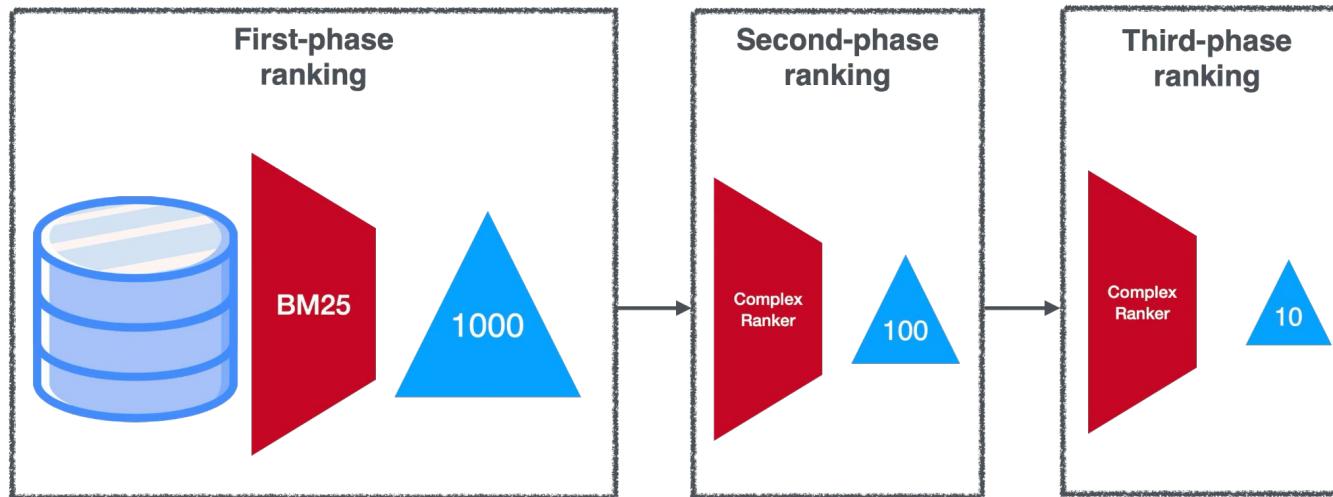


The future of the PISA engine

- We want to build a more user-friendly platform.
- Seamless notebooks – one-command Colab template.
- First-class LSR support.
- MCP server – lightweight micro-control-plane exposing REST/gRPC endpoints for search & index-management
- AI hooks for RAG: flexible connectors that integrate embedding, hybrid retrieval, and LLM post-processing.

Cascading Retrieval

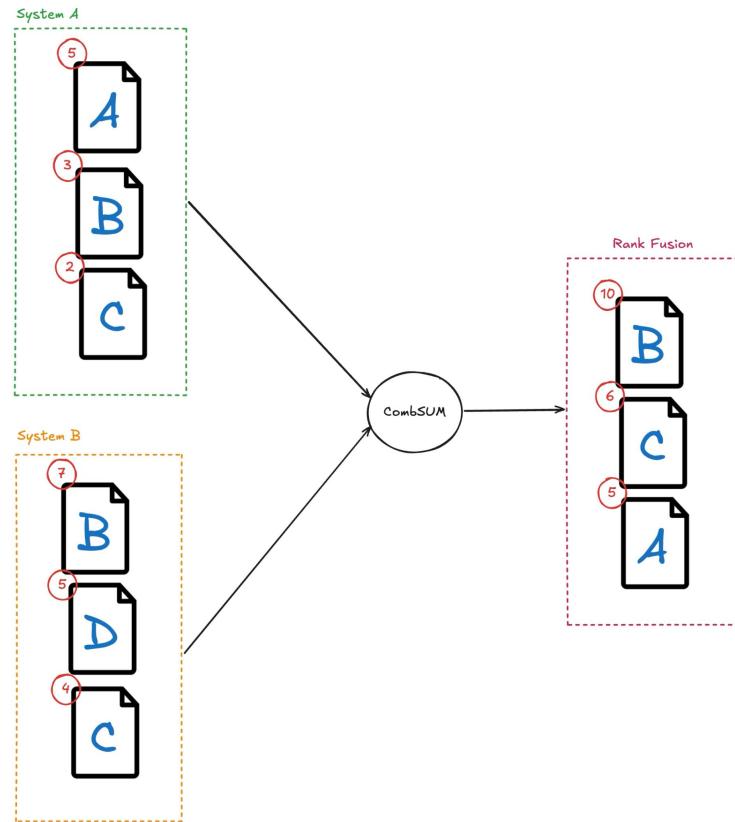
Build a multi-level architecture, from simple to complex (= cheap to expensive)



Hybrid Retrieval

Perform hybrid retrieval via rank fusion mechanisms

Fusion for Information Retrieval is the process of combining multiple sources of information to produce a single result list in response to a query.



Reranking

Two-stage retrieval process inverted index-based candidate generation and multi-vector reranking.

Table 3. Effectiveness metrics and mean response time (MRT, in ms) for top-10 retrieval using PLAID vs. two-stage on Dev Queries, TREC 2019, and TREC 2020.

| | Dev | | TREC 2019 | | TREC 2020 | |
|---------------------------|-------|-------|-----------|-------|-----------|-------|
| | MRR | MRT | nDCG@10 | MRT | nDCG@10 | MRT |
| CoLBERT | 39.99 | 51.25 | 74.26 | 51.46 | 73.99 | 50.21 |
| ESPLADE | 38.75 | 3.07 | 71.33 | 3.13 | 71.14 | 3.20 |
| + ConstBERT ₃₂ | 39.52 | 4.95 | 74.38 | 5.50 | 74.33 | 5.23 |