Implementing Block-Max Pruning in Rust

Faster Learned Sparse Retrieval for Modern Search

1 February 2026

FOSDEM

Team

About us

Antonio Mallia

Founder, CEO

linkedin.com/in/antoniomallia

antonio@seltz.ai

Ferdinand Schlatt

Research Engineer

linkedin.com/in/ferdinand-schlatt

ferdi@seltz.ai

About

Who we are

Seltz

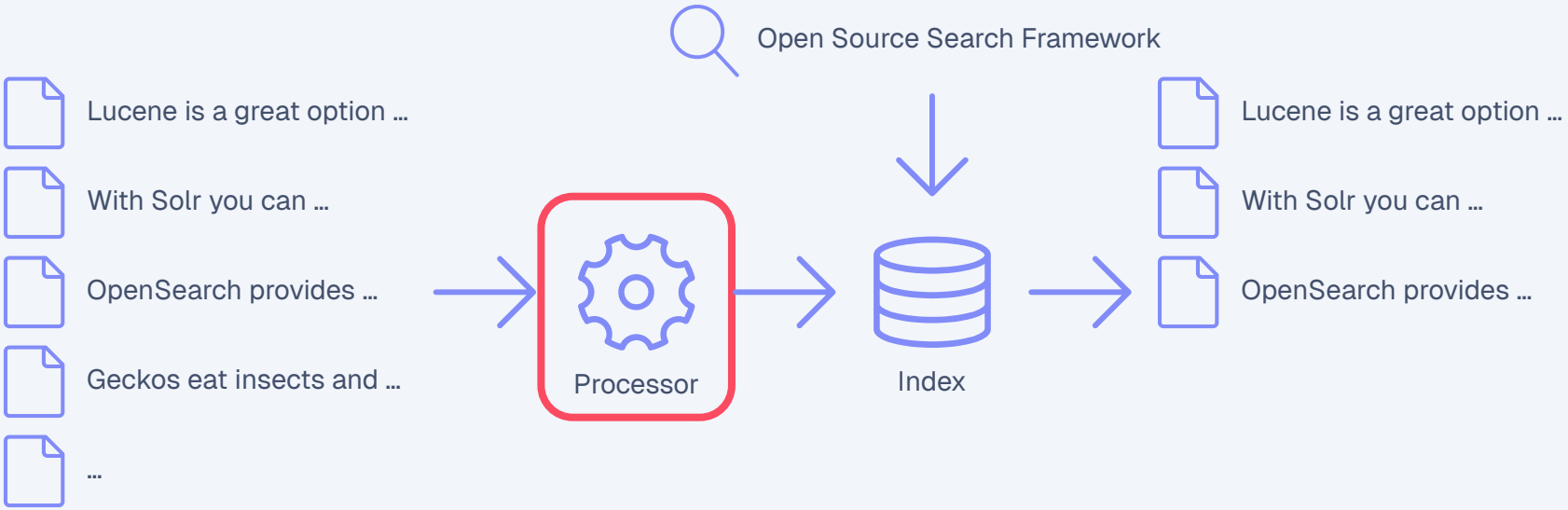
Seltz makes Web knowledge instantly usable for AI.

Founded in 2025, Seltz is designed from the ground up to serve AI as the primary consumer of the Web, providing reliable, efficient, and machine-ready access to web knowledge.

Rather than wrapping third-party services, Seltz owns and operates the entire knowledge pipeline, enabling precise retrieval, clear operational control, and predictable system behavior.

The platform is grounded in research and academia, drawing on advances in information retrieval and systems engineering, and treats efficiency and cost as first-class constraints—delivering scalable, sustainable performance for real-world AI systems.

Ad-Hoc Search



Lexical Retrieval Models

Processor

Lucene is a great option ...

[lucene: 10, is: 15, a: 35, great: 2, ...]

With Solr you can ...

[with: 5, solr: 12, you: 2, can: 6, ...]

OpenSearch provides ...

[...]

Geckos eat insects and ...

[...]

...

...

Lexical Retrieval Models

Index

[lucene: 10, is: 15, a: 35, great: 2, …]

[with: 5, solr: 12, you: 2, can: 6, …]

[…]

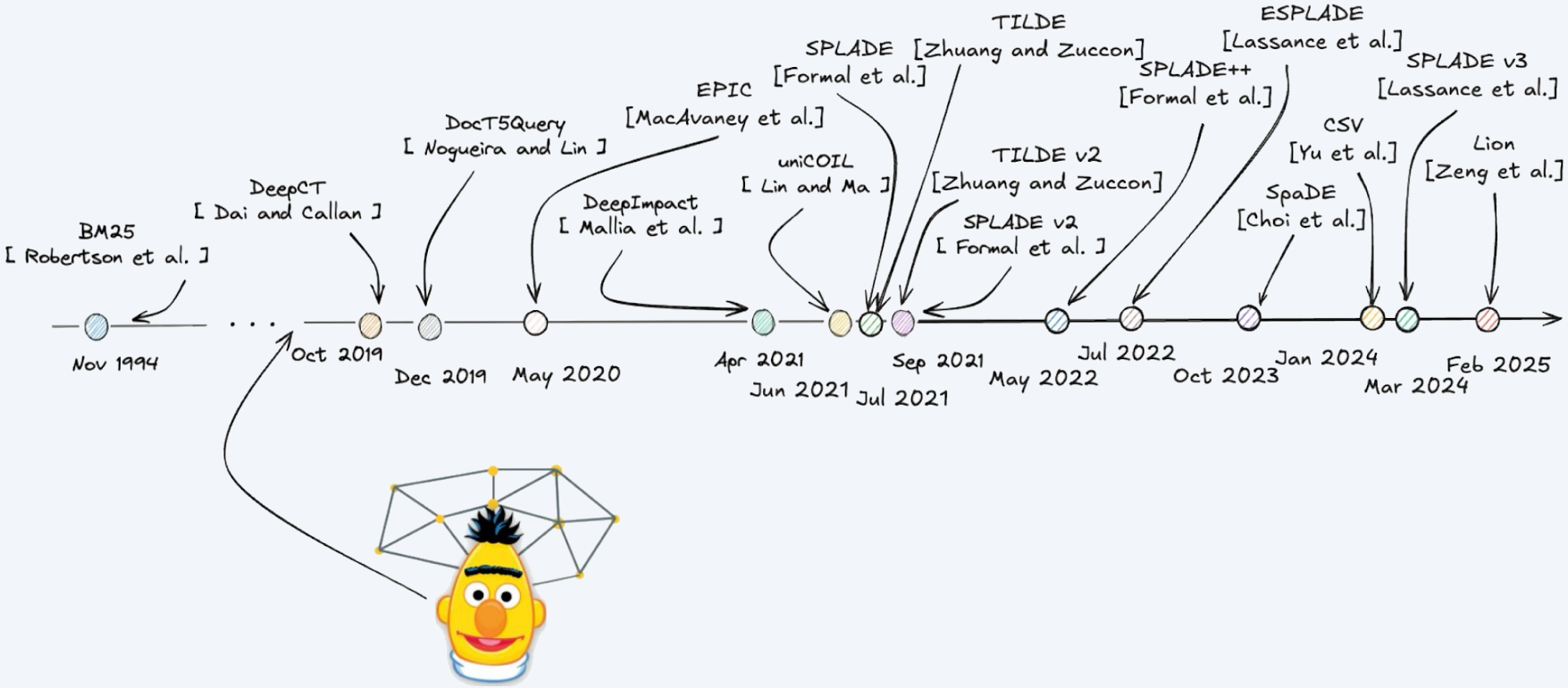
[…]

…

```
lucene: [(Doc_1, 10), (Doc_17, 2), … ]

is: [(Doc_1, 15), (Doc_2, 27), … ]

…
```

- ⊕ Fast indexing & retrieval
- ⊕ Fairly Effective
- ⊖ Only lexical matches
- ⊖ Heuristic scores

History of (Learned) Sparse Models



Sparse Neural Models

Processor

Lucene is a great option ...

With Solr you can ...

OpenSearch provides ...

Geckos eat insects and ...

...

[lucene: 13.5, ~~is: 0.0~~, ~~a: 0.0~~,
great: 0.1, option: 0.2, ... , search: 6.5,
framework: 4.5]

- ⊕ Learnable impact scores
- ⊕ Semantic term expansion
- ⊖ Slow and expensive indexing
- ? Fast retrieval?

“Wacky Weights”

Can we reuse the index structures developed for term statistics?

No. At least the optimizations developed for term frequency distributions do not work well with LSR models.

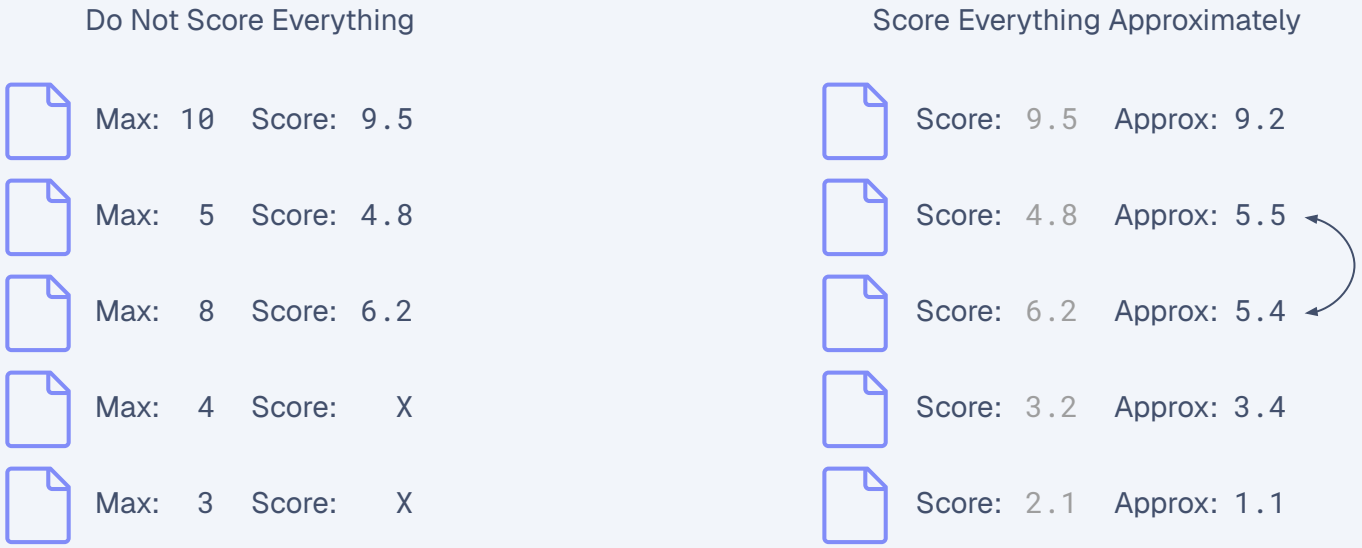
Method	|V|	Avg. Terms/Doc	Average Terms/Query	Latency* (ms)
BM25	2 660 824	30.1	5.8	8.3
DeepImpact	3 514 102	71.1	4.2	19.4
uniCOIL	27 678	66.4	6.6	36.9
SPLADE	28 131	229.4	25.0	220.3

Term distributions in the MS MARCO passage corpus with different processors
*Using the PISA retrieval engine.

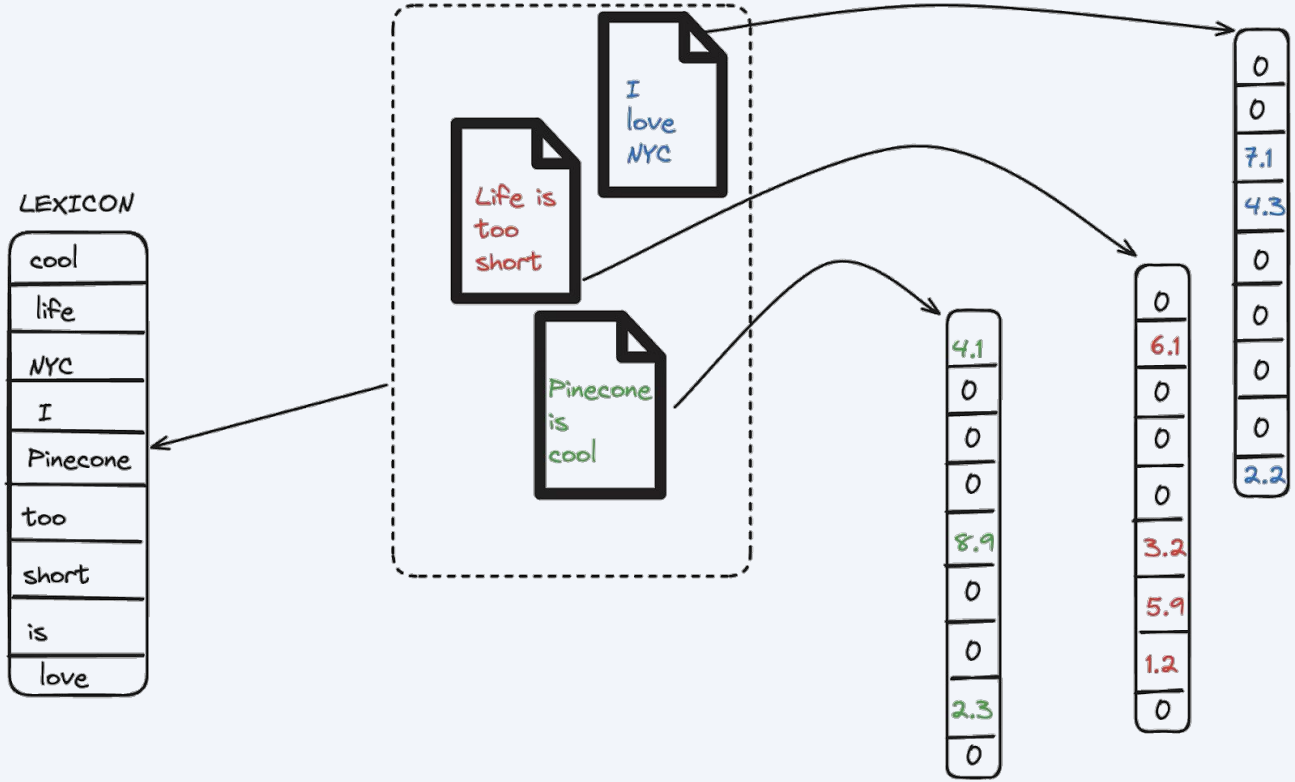
Joel Mackenzie et al., Wacky Weights in Learned Sparse Representations and the Revenge of Score-at-a-Time Query Evaluation, ArXiv 2021

Early Exiting and Approximate Scoring

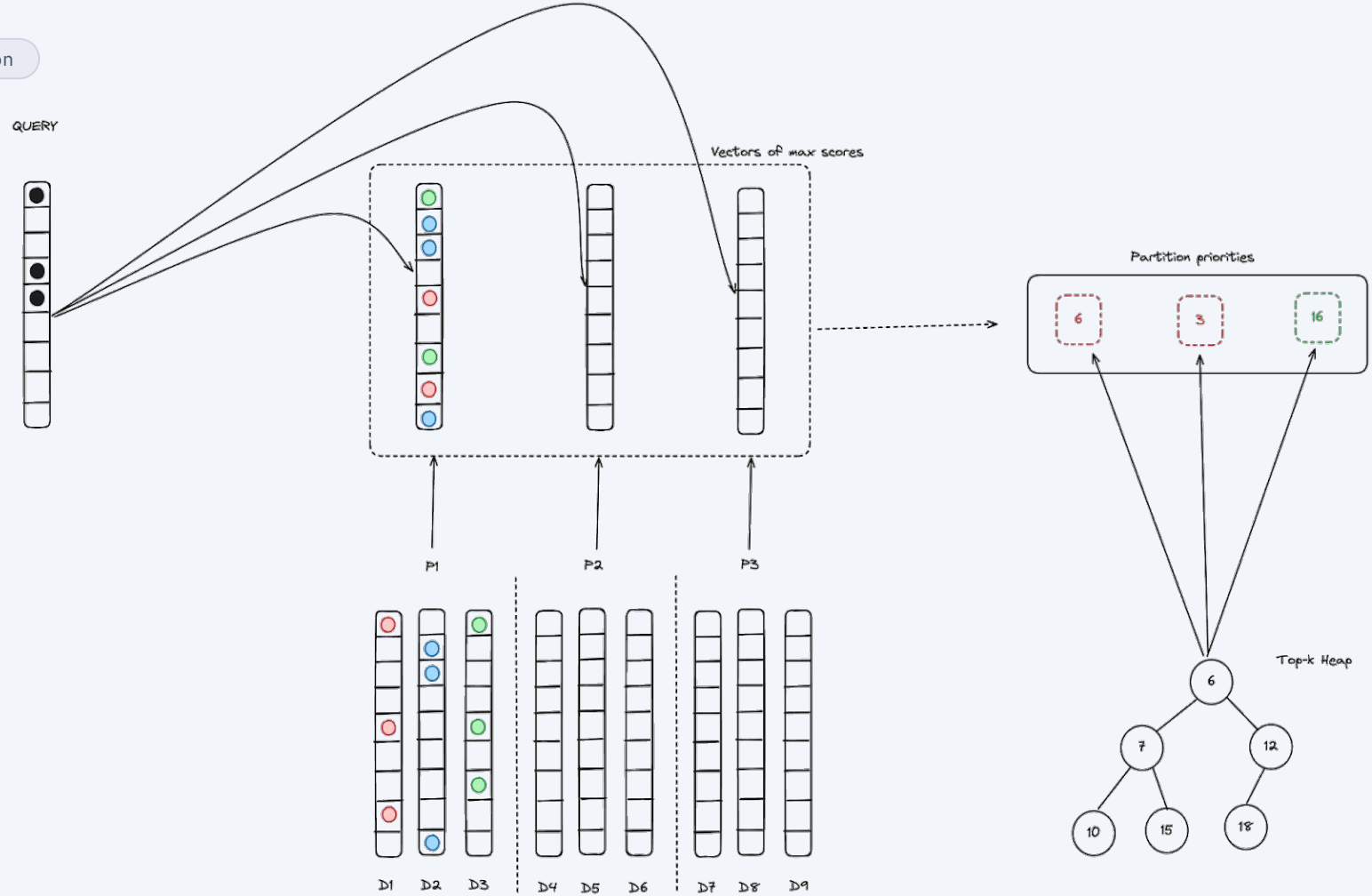
How can we improve the retrieval efficiency for LSR models?



Block Max Pruning



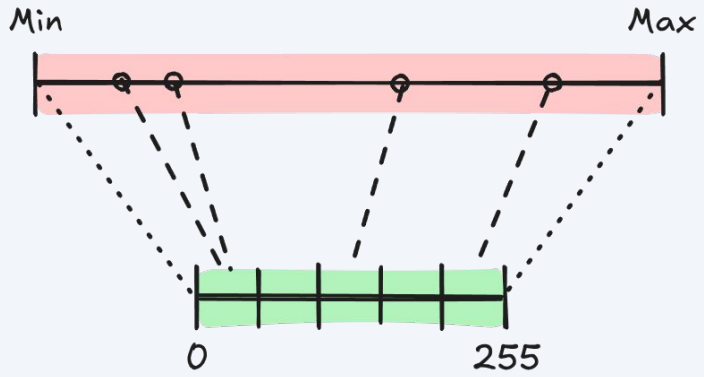
Solution


QUERY
Vectors of max scores
P1
P2
P3
D1
D2
D3
D4
D5
D6
D7
D8
D9
Partition priorities
6
3
16
Top-k Heap
6
7
12
10
15
18

Solution

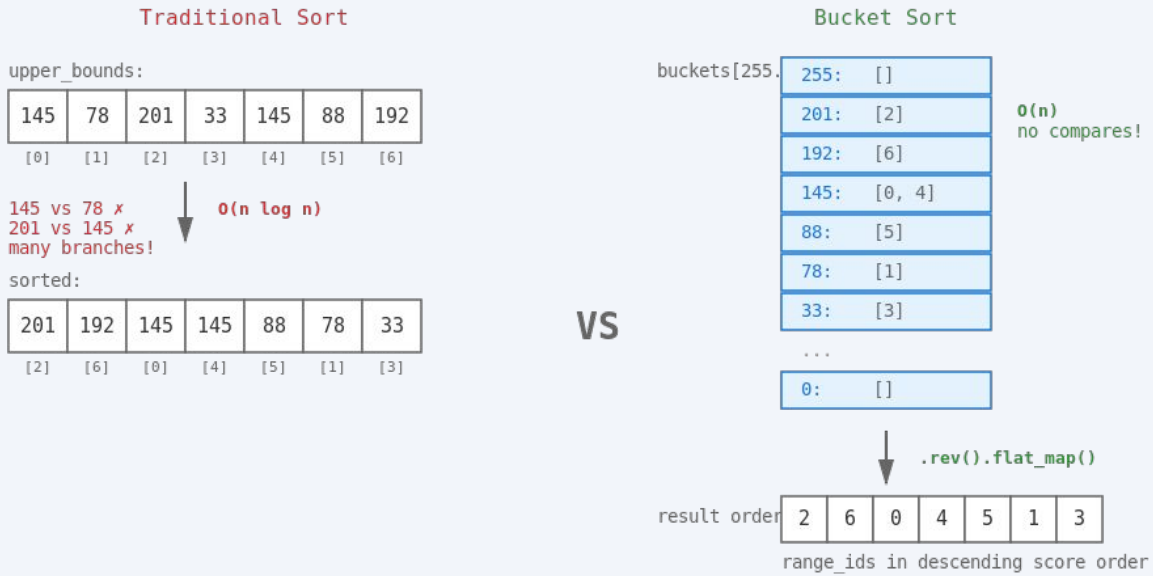
Quantization

- Map float32 values to int8 values
- Map minimum value to 0 and maximum value to 255
- Faster scoring with minimal precision loss



Solution

Traditional Sort vs. Bucket Sort



Key Insight: Since scores are bounded [0, 255], we use the score as a direct array index
Result: Zero comparisons, zero branch mispredictions, perfect for CPU pipeline!

Solution

Zero-Comparison Sorting with Bucket Sort

1. Bucketing: Place each range_id into bucket indexed by its upper bound score

2. Reverse Iteration: Traverse buckets from 255→0, naturally yielding sorted order

3. Lazy Evaluation: flat_map produces ranges on-demand, no need to materialize sorted array

4. Memory Reuse: Clear and reuse buckets across queries to avoid allocations

Real-World Impact:

- Zero branch mispredictions from comparisons

- Better cache locality (sequential bucket writes)

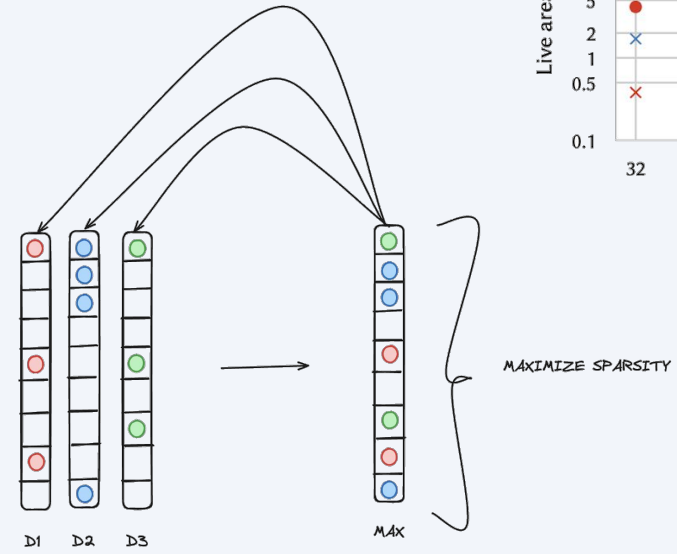
- Allocation-free in query hot path (reuse buckets)

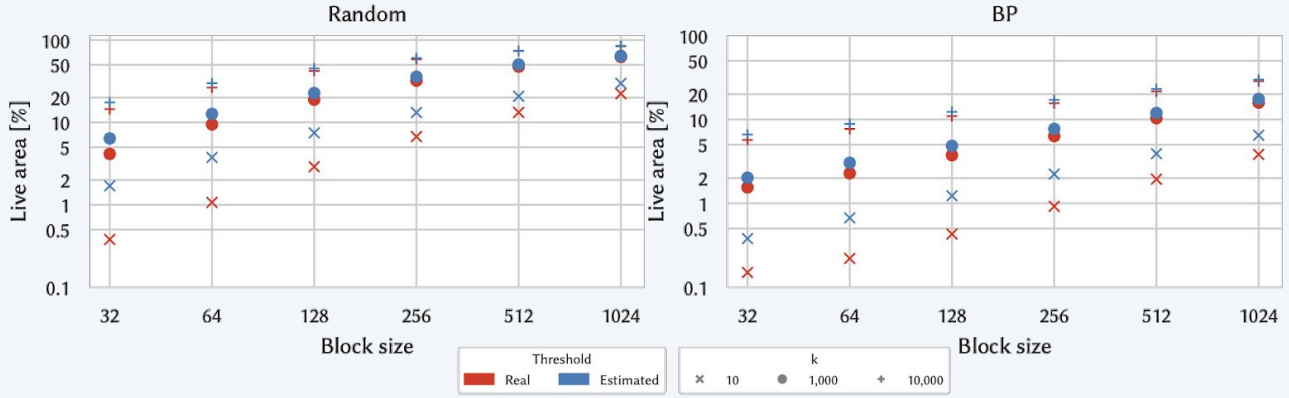
sort.rs

```rust
// Instead of comparison-based sorting: O(n log n)
// upper_bounds.sort_by(|a, b| b.cmp(a));

// Use bucket sort: O(n + k) where k = max value
let mut buckets: Vec<Vec<u32>> = vec![Vec::new(); 2usize.pow(16)];

// Clear buckets for reuse (avoid allocations!)
buckets.iter_mut().for_each(Vec::clear);

// Distribute ranges into buckets by their upper bound score
upper_bounds.iter().enumerate().for_each(|(range_id, &ub)| {
    if ub > threshold {
        buckets[ub as usize].push(range_id as u32);
    }
});

// Iterate in descending order without sorting!
let mut ub_iter = buckets
    .iter()
    .enumerate()
    .rev()  // Start from bucket 255, 254, 253...
    .flat_map(|(score, ranges)| {
        ranges.iter().map(move |&range_id| (score, range_id))
    });
```

Solution

Clustering of Documents





Comparison to Baselines

Method	k=10	k=100	k=1000
MaxScore	120.6	152.8	193.8
BMW	614.2	658.7	686.7
BMP	11.0	15.9	37.8

Comparison to Baselines

Method	k=10	k=100	k=1000
MaxScore	120.6	152.8	193.8
BMW	614.2	658.7	686.7
IOQP	79.1	80.2	80.8
Anytime	80.6	114.0	163.1
Clipping	245.9	358.8	504.1
BMP	11.0	15.9	37.8

Context

Top-k Heap

The Insight:

- The **root of a min-heap** contains the **smallest** element in the heap

- For top-k, this root is the **k-th best score** (threshold)

- Any new score ≤ threshold can be **rejected in O(1)** without touching the heap

- Any new score > threshold **replaces** the minimum (evicts the worst)

// Storing all documents

let mut all_docs = Vec::new(); // Size: O(n) - millions of entries!

all_docs.sort(); // Time: O(n log n)

let top_k = &all_docs[..k];

heap.rs

```rust
pub struct TopKHeap<S> {
  heap: BinaryHeap<Entry<S>>,  // Min-heap via reversed ordering
  threshold: S,                // k-th best score (heap root)
  k: usize,
}

impl<S: Default + Copy + PartialOrd> TopKHeap<S> {
  /// Fast O(1) pruning check
  pub fn would_enter(&self, score: S) -> bool {
    score > self.threshold  // Compare against k-th best
  }

  /// Insert with automatic threshold update
  pub fn insert(&mut self, doc_id: DocId, score: S) -> Option<S> {
    // Early rejection - no heap operations needed!
    if !self.would_enter(score) {
      return None;
    }

    // Accept: push new entry
    self.heap.push(Entry { doc_id, score });

    // If heap exceeds k, evict minimum
    if self.heap.len() > self.k {
      self.heap.pop();  // Remove worst entry
      self.threshold = self.heap.peek().unwrap().score;  // Update threshold
      Some(self.threshold)
    } else if self.heap.len() == self.k {
      // Heap just filled up, set initial threshold
      self.threshold = self.heap.peek().unwrap().score;
      Some(self.threshold)
    } else {
      None
    }
  }
}

// The clever trick: reverse comparison to make max-heap act as min-heap
impl<S: Copy + PartialOrd> Ord for Entry<S> {
  fn cmp(&self, other: &Self) -> Ordering {
    // Note: other vs self (reversed!)
    other.score.partial_cmp(&self.score)
      .unwrap_or_else(|| other.doc_id.cmp(&self.doc_id))
  }
}
```

Usage

Using BMP

- Rust and Python interface
- Batch and single query interface
- Easy index creation using CIFF

bmp.py

```python
from bmp import search, Searcher

# batch interface
results = search(
   index="/path/to/index",
   queries="/path/to/queries",
   k=10,
   alpha=1.0,
   beta=1.0,
)
# single query interface
searcher = Searcher("/path/to/index")
results = searcher.search(
   {"tok1": 5.3, "tok2": 1.1},
   k=10,
   alpha=1.0,
   beta=1.0,
)
```

Usage

CIFF

- Common Index File Format
- Defined as part of the Open-Source IR Replicability Challenge (OSIRRC)
- We built tools to convert between different formats
  - CIFF ←→ PISA
  - JSONL → CIFF
  - CIFF → BMP

Jimmy Lin et al. Supporting Interoperability Between Open-Source Search Engines with the Common Index File Format. SIGIR 2020.

ciff.proto

```
message Header {
    ...
}

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 {
    ...
}
```

CIFF

- Common Index File Format
- Defined as part of the Open-Source IR Replicability Challenge (OSIRRC)
- We built tools to convert between different formats
  - CIFF ←→ PISA
  - JSONL → CIFF
  - CIFF → BMP

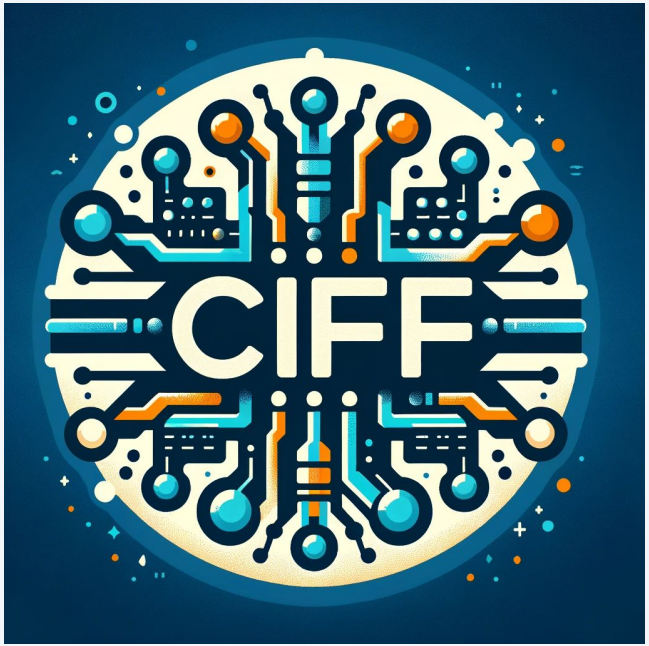
```
$ ciff2bmp -c index.ciff -o index.bmp
```

Jimmy Lin et al. Supporting Interoperability Between Open-Source Search Engines with the Common Index File Format. SIGIR 2020.

CIFF Hub

- Repository of CIFF indexes (and queries) for several popular models and corpora
- Download an index and easily compare different retrieval engines

pisa-engine/ciff-hub



Further Reading

- Mallia, Antonio, et al. Faster learned sparse retrieval with block-max pruning, SIGIR 2024
- Parker Carlson, et al. Dynamic Superblock Pruning for Fast Learned Sparse Retrieval, SIGIR 2025
- Yifan Qiao, et al. Threshold-driven Pruning with Segmented Maximum Term Weights for Approximate Cluster-based Sparse Retrieval, EMNLP 2024
- Joel Mackenzie, et al. Efficient In-Memory Inverted Indexes Theory and Practice, SIGIR 2025

Demo

Thanks!

pisa-engine/BMP

Presented By

Seltz