Lucene Cyborg: Hybrid Search Engine Written in Java and C++

Doo Yong Kim

E–mail: *0ctopus13prime@gmail.com*

SUMMARY: Apache Lucene, the popular search engine library written in Java, has maintained its prominence for over 25 years. Despite its extensive usage, there have been numerous efforts to improve its performance by exploring implementations in compiled languages. However, the literature lacks clear empirical evidence on the performance benefits of such porting, leaving users to rely on speculative assessments. This paper introduces the concept and design of LuceneCyborg, a hybrid approach where Java delegates search requests to C++ for potentially enhanced performance. This paper aims to quantify the performance improvements possible with this design. The benchmarks focused on measuring execution times for conjunction and disjunction query types across three command types: TOP_100, TOP_100_COUNT, and COUNT. The results demonstrate that LuceneCyborg consistently outperforms baseline for all types of conjunction queries. Specifically, LuceneCyborg exhibited a performance improvement of 26.9% for COUNT-Conjunction queries and up to 27.7% for TOP_100-Conjunction queries. Additionally, it was 17.57% faster for TOP_100-Disjunction queries. However, LuceneCyborg's performance was slower for disjunction queries: it was 4% ˜ 32% slower than the baseline for COUNT, TOP_100_COUNT Disjunction queries.

Key words. Search Engine, Open Source System

1. INTRODUCTION

Apache Lucene, initially developed in 1999, has become the de facto standard search engines, serving as the backbone for numerous search infrastructures over the past 25 years. Despite its widespread adoption and continuous evolution, there have been numerous efforts to further increase its search performance by rewriting it in compiled languages. These attempts aim to leverage the inherent efficiency of compiled languages over Java Virtual Machine (JVM), which Lucene runs on.

However, there is a notable gap in the literature: no comprehensive review or analysis exists that compares the advantages and disadvantages of these rewritten Lucene alternatives against the original Java-based Lucene. Michael McCandless, a PMC of Lucene, conducted experiments in this area. He attempted to delegate the search processing to C++ through Java Native Interface (JNI), and his findings indicated a search performance improvement of up to 300-400% across various query sets.

While McCandless's work highlighted potential performance gains, there have been massive advancements in both Lucene's internal codecs and JVM since his experiments eleven years ago. These make it difficult to accurately compare the performance of Lucene with potential C++ implementations, leading to speculation rather than concrete conclusions about the benefits of rewriting Lucene in C++.

Therefore, this paper is aiming to give an overview of Lucene Cyborg, a C++ reimplementation of Apache Lucene, and evaluates its performance benefits across various text queries. In Section 1, I revisit lessons learned from an earlier project that attempted to rewrite Lucene in C++. These insights are continuously developed in Section 2 to present the core design principles of Lucene Cyborg. In Section 3, then present benchmark results comparing Lucene Cyborg to other search engines like Tantivy and Pisa, analyzing its performance. Finally, we summarize the key findings from these benchmarks and propose areas for future research in Section 4.

1.1. Lucene

Lucene, a Java based search library with over 25 years of history, was originally developed to support text indexing and search functionalities. Over time, its capabilities have expanded gradually. Lucene now supports indexing and searching of geographical point types with up to eight dimensions at maximum. Additionally, it offers advanced features for indexing and performing approximate nearest neighbor (ANN) searches on high-dimensional vectors.

Lucene's rich features and high performance have established it as the de facto standard in search libraries. It underpins major platforms like Elasticsearch and Solr, which are widely used to manage search queries in numerous corporations.

From a text searching perspective, Lucene uses Block-Max-WAND algorithm [Shuai Ding (2011)] to efficiently skip non-relevant documents, significantly enhancing performance. Since its integration, this algorithm has improved disjunctive search speeds by 5 to 8 times, marking a substantial advancement in search efficiency [Grand (2020)].

1.2. Search engines written in compiled language

Performant Indexes and Search for Academia (PISA) is an experimental search engine designed to explore efficient implementations of state-of-the-art representations and algorithms in text retrieval [Mallia et al. (2020)]. Written in C++, PISA avoids runtime virtual calls in favor of static polymorphism via C++ templates and metaprogramming, optimizing performance. To enhance search speed, it reorders documents during index construction. Additionally, PISA supports the Block-Max WAND algorithm, further boosting its search efficiency.

Tantivy is a search library developed in Rust. It proclaims its design was highly inspired Lucene. [Masurel (2016)] It shares a similar architectural approach with Lucene, notably the use of immutable segments that are incrementally merged based on a defined merge policy. These segments are then loaded into memory using the mmap system call. Tantivy also employs finite-state transducers to construct its term dictionary [phiresky (2021)], in a similar way that Lucene builds its own dictionary.

1.3. Past experience

Through my previous projects over the past few years, I have gained valuable insights. [Kim (2019, 2020)] The project aimed at transplanting Lucene into C++ to minimize search latency, before Block-Max-Wand was available in the Lucene ecosystem (in the end, the project was paused). Below are key lessons distilled from this experience:

Lesson 1. C++ exhibits stronger and more stable computational performance compared to Java: For tasks such as floating-point ranking score calculations or bulk encoding and decoding, C++ consistently demonstrates better stability with smaller latency deviations than Java. As of the latest benchmarks, although the performance gap has narrowed to nearly 20% for intensive computational algorithms [benchmarksgame (2024)], C++ still tends to provide more predictable latency patterns, whereas Java occasionally experiences spikes in its 99th percentile latency (p99). However, achieving stable and superior performance in C++ requires significant time invested in constant metrics monitoring and experimentations. Eventually, keeping this diligent process leads to surpassing the JVM's performance, although the path to achieving this milestone is challenging and demanding.

Lesson 2. Single Instruction/Multiple Data (SIMD) instruction [Wikipedia-SIMD (2024)] is important but not the critical factor influencing text searching latency: The critical factor affecting latency lies in reducing the search space (i.e. the number of documents to process). While SIMD can contribute to performance gains, its impact was typically marginal. In many cases, poorly optimized implementations may actually result in slower performance, especially when CPU underlying interactions are not carefully considered. Thus, effective management of the search space remains critical for optimizing latency in computational tasks.

Lesson 3. Expanding on the previous point, achieving optimal performance could be achieved by integrating the WAND algorithm [Broder (2003)] with wise strategies that maximize L1 and L2 cache hit ratios effectively. We can still achieve fast performance even with general file formats that do not require AVX2 SIMD instructions [Wikipedia-AVX (2024)] at all.

Lesson 4. Compile time polymorphism is powerful: Specifically, it involves function calls that are resolved during compilation using templates. Providing clear type information through templates enables the compiler to generate efficient machine code, incorporating optimizations like inlining and code rearrangement.

For instance, Lucene's Boolean query [Lucene-BooleanQuery (2024)] can be a good candidate for applying compile time polymorphism. In which, inner nodes representing ”OR” or ”AND” operators are instantiated with actual query types during compilation. This contrasts with Lucene's approach, which relies on virtual functions in C++ for query processing.

On the flip side, adding a new type to the boolean query tree necessitates recompilation, potentially leading to decreased productivity due to longer testing and deployment times.

Lesson 5. The search performance of BKD trees is comparable between C++ and Java: A BKD tree [Lucene-BKDWriter (2024)] is a variant of the KD tree that stores document IDs in leaf nodes, with internal nodes containing dimension and splitting value bytes. In Lucene, it opts for binary trees when indexing one-dimensional data; otherwise, BKD trees are employed. (conceptually, BKD tree is a super set of a binary search tree) Therefore, the most time-consuming task involves traversing block document IDs, and the time spent on index tree operations occupies a minimal portion.

Lesson 6. Decompression performance for stored fields (using LZ4 or Gzip) is comparable between C++ and Java, assuming that C++ employs the same algorithm as used in Lucene (note that there are multiple variants of LZ4). Stored fields in Lucene offer multiple compression options, with LZ4 being the default choice. In the current implementation of Lucene, data is buffered until it either exceeds 80KB or the number of stored documents reaches 1024, after which compression is applied to the buffered data. LZ4 decompression primarily involves three operations [Lucene-LZ4 (2024)]:

1. Shift operator: In LZ4, length information is stored in 4 bits. Interpreting a one-nibble token requires the use of the left shift operator.

2. Loop memory copy: When an end offset of matching contents extends beyond the current file offset, manual looping is required to copy overlapping matched content into a read buffer. This scenario commonly occurs when there are many consecutives bytes.

3. memcpy: When the matching content is already within a decompressed chunk (where the end offset is less than the current offset), data can be efficiently loaded into a read buffer using a simple memcpy operation. Note that in Java, System.arraycopy internally invokes native memcpy.

Because of this characteristic, which enables decompression using simple operations and native memcpy calls, I have not found compelling evidence that rewriting Lucene's LZ4 algorithm in C++ would result in superior decompression performance.

Lesson 7. In terms of index construction, C++ rewritten version demonstrated comparable write throughput to Java. While C++ rewritten version experienced less memory pressure without the overhead of garbage collection (GC), it is required additional validation to determine potential upper limits of memory conservation. However, I believe that addressing memory pressure issues should leverage recent advance Java's low-level API features for managing native allocated memory, rather than relying on C++. In addition to equivalent write throughput, another reason why I consider C++ less suitable for index construction is the significantly challenging maintenance it entails. Lucene's IndexWriter heavily relies on maximizing parallelism, leveraging the strong guarantees provided by the JVM specification. IndexWriter class in Lucene includes several critical data structures for managing current segment information or flushed segments awaiting publication. Multiple worker and merge threads concurrently access these structures to add delete requests to a lockless volatile list [Lucene-DocumentsWriterDeleteQueue (2024)] (here, 'volatile' is defined as per Java standards, whose definition is different to the one in C++ language spec) Three challenges emerged when using C++ ported IndexWriter.

1. Difficulties in pinpointing the root cause of failures: In C++, all allocated resources must be meticulously managed, making it extremely difficult to trace issues like segmentation faults, often being raised from dangling pointer access violations. This complexity is compounded by non-deterministic nature of Lucene's index construction mechanism, where worker threads handle pending tasks in a unpredictable order (except when it is explicitly configured to use a single thread). Moreover, after spending considerable time stabilizing the system, segmentation fault issues became challenging to reproduce, often surfacing only after indexing a large number of documents — such as 50 million. Identifying the root cause of these issues typically required several days or even weeks of thorough investigation.

2. Preventing circular references: While circular reference are less concerning issues in Java world, they can lead to memory leaks in C++ unless carefully managed, particularly with shared pointers (std::shared_ptr). I discovered that even after deallocating IndexWriter instance in C++, few data structures remained alive due to circular references. Such issues would have been automatically cleaned up within JVM. During Lucene indexing, data structures or meta-information are passed between several objects, which then inject these parameters into others. This creates a complex object graph rather than a straightforward tree topology, making it challenging to design a concise solution for determining resource cleanup with std::shared_ptr in C++. Additionally, identifying whether a data structure is still in use by a worker thread added further complexity in deciding the correct order for resource cleanup. JVM ensures that unused resources are properly cleaned up, whereas in C++, this responsibility falls entirely on the author.

3. Combining the aforementioned points, I observed that nearly all colleagues struggled to fully understand how the code functions, resulting in a linear increase in communication costs over time. Without a solid grasp of the code's workings, implementing new features on top of it required extensive alignment

and testing periods.

Lesson 8. In terms of memory allocation, Java holds an advantage over C++. Java's memory allocation is faster than C++'s 'new' operator. Simply porting Lucene's code to C++ could degrade performance due to the overhead involved in object allocations per search query processing. For instance, consider the minimum number of objects listed below that need to be instantiated to perform a text query per segment. 1. IndexSearcher 2. Query 3. Collector 4. Weight : Note that several Weight objects can be made during query rewriting step. 5. BulkScorer 6. ScorerSupplier 7. Scorer : Lucene constructs a scorer tree where a parent scorer has child scorers. 8. Iterator : Likewise Scorer, Lucene maintains an Iterator tree inside. 9. List, Map and Queue data structures. 10. TermsEnum 11. PostingsEnum or ImpactsEnum 12. LeafSimScorer 13. Other internal read buffers to decode FST, suffix Trie and posting list.

Therefore, without a carefully designed approach to object allocations during search operations in C++, the time spent on allocation and deallocation could significantly increase latency and impact search performance. This issue was also evident in the LuceneCPlusPlus project [Lucene++-Issue178 (2021)], where the construction and destruction of std::shared_pt resulted in a 256% increase in latency compared to the baseline Lucene 3.0.3. In C++, memory allocation needs synchronization due to the potential for memory allocated by one thread to be released by another. This mutual protection of internal memory regions against simultaneous updates can degrade performance under heavy search traffic. One approach to mitigate this issue is using jemalloc [Evans (2006)], which distributes traffic to reduce contention per memory arena between threads, though it still involves locking. In Lucene Cyborg, I addressed this issue by introducing a local linear allocator in each thread to minimize locking.

1.4. Lucene Cyborg

"Cyborg" is a term that describes a cybernetic organism, which enhances human physiological functions with machine technology. Likewise, Lucene Cyborg internally uses a heterogeneous system combining Java and C++ to achieve the same search functionality as Lucene. This is accomplished by delegating search requests to a shared library written in C++ via Java Native Interface (JNI) within the JVM. On the Java side, it validates input and prepares necessary information to pass downstream. The entire search process, including navigation of inverted indexes and decoding of posting lists, is executed by machine code compiled from the C++ source.

1.4.1. Goal

The principal tenet of Lucene Cyborg is to implement Lucene's algorithms in an efficient manner, enabling it to operate on indexes built by Java Lucene with improved performance. From the C++ perspective, Lucene Cyborg relies on the equivalent search algorithms used by Lucene to achieve identical search results. Internally, it attempts to optimize implementations by leveraging various C++ optimization techniques to enhance performance. It does not simply replicate Java source code in C++; instead, it explores multiple approaches to reduce memory allocation and deallocation overhead. It also provides explicit type information to the compiler, aiming to generate as efficient machine code as feasible. The main advantages of this design decision, as opposed to creating a new algorithm with custom file formats, can be summarized in three points.

1. It provides much easier readability for Lucene users. Anyone familiar with Lucene can easily test and debug Lucene Cyborg since it shares the same conceptual framework.

2. Expanding on the first bullet point, Lucene Cyborg naturally benefits indirectly from ongoing improvements made by Lucene committers to its established algorithms and file formats.
The design decision to make Lucene Cyborg compatible with Lucene allows it to seamlessly integrate new enhancements with minimal effort.

3. As a result, maintaining a high standard of quality requires relatively less effort, leveraging improvements driven by the collective intelligence of the open-source community. In contrast, I've observed that many native search library projects in the past have become archived or nearly stalled due to challenges in obtaining volunteer contributions across various aspects.

For example, Lucene supports indexing of point types using BKD trees. If I were trying to introduce a similar indexing data structure to its own library, it would involve a longer alignment phase during initial design, not to mention extensive testing and code reviews.

2. Methods

In this section, I propose the design principles of Lucene Cyborg, reflecting lessons learned from past projects.

2.1. Scope

My approach does not aim to achieve performance gains by exploring new algorithms, restructuring underlying file formats, or applying different encoding/decoding schemes from those used in Lucene. As discussed in Section Lesson 1, Lucene Cyborg approach solely focuses on implementation, leveraging C++ language features to enhance performance while adhering to identical algorithms, file formats, and encoding/decoding schemes used in Lucene. Additionally, benchmark experiments covered in this paper focus on text boolean search. Experiments involving vector approximate nearest neighbor (ANN) search and other types of searching will be conducted in future rounds of experimentation.

2.2. Lucene Cyborg Design

Lucene Cyborg focuses solely on searching and does not include index construction. This decision reflects the lesson 7 from past projects, where rewriting index construction in C++ did not yield benefits. Lucene Cyborg guarantees to produce identical results as Java Lucene when given the same input.

2.2.1. High level design

At a high-level overview, two packages interact to achieve search functionalities:

1. Lucene Cyborg Java [Kim (2024b)]: Pure Java implementation that validates inputs and passes required information to Lucene Cyborg C++.

2. Lucene Cyborg C++ [Kim (2024a)]: Written in C++, implements Lucene's decoding and search algorithms.

I will explicitly refer to "Java Lucene" when discussing the baseline Java implementation to minimize confusion.

In Lucene Cyborg's Java side, it handles input validation and prepares necessary information for searching, such as index path, field name, and text term. This information is then passed to the shared library (Lucene Cyborg C++). This search functionality only becomes available after loading the shared library into JVM.

After loading the library into the JVM (e.g. System.load), Lucene Cyborg Java passes the index path via JNI to delegate the initialization phase to Lucene Cyborg C++. Here, it maps Lucene segment files into memory using the system call mmap to prepare them for subsequent access. Similar to Java Lucene, during initialization, Lucene Cyborg C++ performs tasks such as codec header CRC checks.

On the Lucene Cyborg C++ side, it receives information via JNI from Lucene Cyborg Java to execute searches. After completing the search, it decorates the obtained results to Java instances using JNI C++ API.

2.2.2. Low level design – Lucene Cyborg C++

In this section, I outline low level design details in Lucene Cyborg C++.

1. Using the mmap system call is the sole option available to load the index into memory. Lucene offers various options for loading indexes into memory, including explicitly locating regions on disk using fseek system call and selectively copying them into memory.

2. Lucene Cyborg C++ ensures that logical index regions are mapped continuously in memory. There are several strategies for mapping index files using mmap. A naive approach is to map the entire file into memory, but this method increases loading time significantly when dealing with large index files exceeding 5 GB, as observed in my previous project. Another approach is to mmap the index file in fixed-size multiples of the page size (e.g., 16 KB). However, this method does not consider logical regions where one region might span across two mmapped areas, necessitating a read buffer to combine the two parts.

For example, Lucene employs two trie data structures: the first trie indexes prefix strings (tip file), while the second trie, known as the suffix trie, stores suffix strings (tim file). Lucene attempts to bundle multiple suffix strings together in a single block. However, when many strings sharing a common prefix resulted in too many suffix strings, it uses several blocks (called floor blocks) to accommodate them. It saves the first character (lead byte) of the first string in every block so that it can skip unnecessary floor block scan via binary search during searching.

Therefore, the end offset of the last floor block of suffix trie is the key to boundary calculation. (The end offset automatically becomes the start offset of next field). By mmaping logical region into memory, Lucene Cyborg C++ can directly access with offset without having to copy it into a read buffer.

3. LinearAllocator : All memory allocations during search are enforced to be allocated from locally created LinearAllocator. This principle reflects the lesson 8 in Section 1, where frequent memory allocations and deallocations degrade performance.
LinearAllocator simply allocates a 64KB memory chunk internally and maintains an offset that represents the last position of the used area within the chunk. When a memory allocation is requested, it merely increments the offset and returns the starting offset. If there's no free space left in the chunk, it allocates a new chunk of the same size and provides the requested memory from there. When deallocation is requested, the LinearAllocator performs no immediate operation. Instead, it releases allocated memory chunks in its destructor eventually.

With this approach, memory allocation becomes

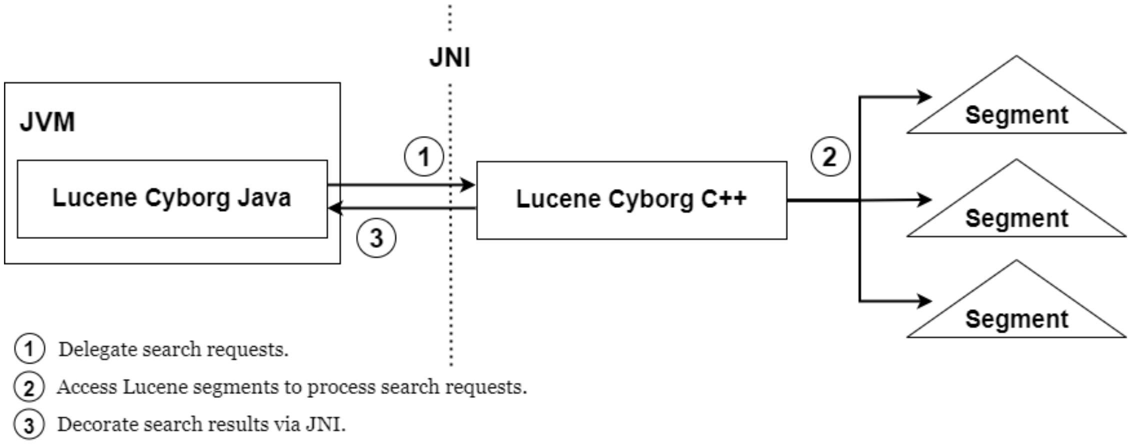
JNI
JVM
Lucene Cyborg Java
Lucene Cyborg C++
1
2
3
Segment
Segment
Segment
1 Delegate search requests.
2 Access Lucene segments to process search requests.
3 Decorate search results via JNI.

Fig. 1: High-level architecture of Lucene Cyborg: Lucene Cyborg Java prepares input running in the JVM, delegates search requests to a shared library written in C++ (denoted as Lucene Cyborg C++). Lucene Cyborg C++ then iterates through the inverted index for each Lucene segment to proceed with the search. After searching, it then decorates the results using JNI and gives control back to the JVM.

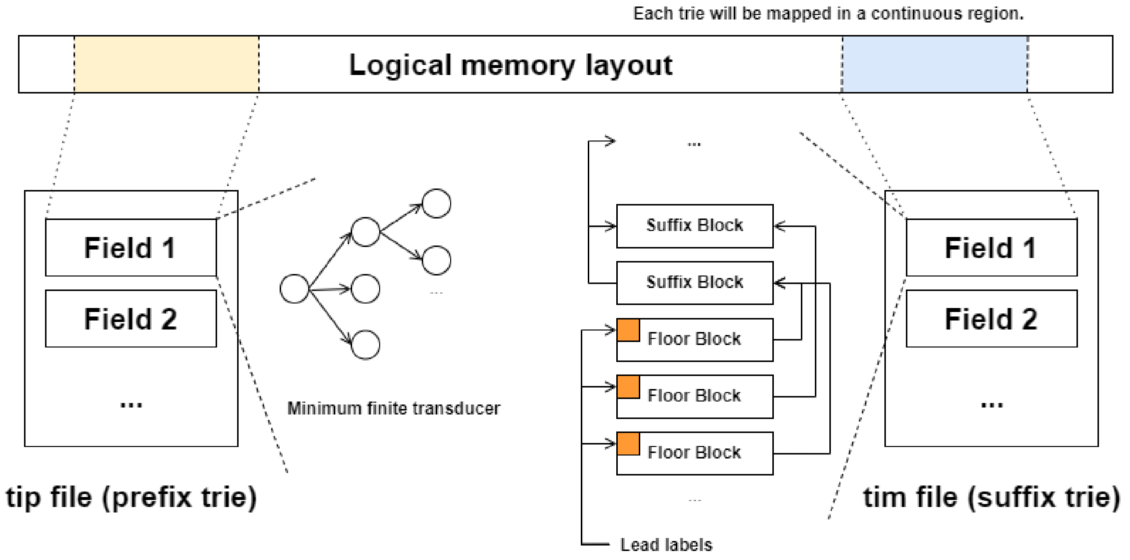

Fig. 2: In Lucene Cyborg, it tries to mmap the each data structure in a continuous region. In this example, it mmaps the entire prefix trie, where internally Lucene uses a minimum finite transducer. And it does the same mapping to suffix tries as well.

as simple as incrementing an integer variable, without needing to lock the region for mutual exclusion. It is a valid argument that the number of malloc calls are bounded by a constant factor, because typically, one search transaction per a segment rarely requires more than 64KB of memory space.

4. Pre Compiled Optimization (PCO) : Using C++ templates, Lucene Cyborg precompiles possible combinations to minimize virtual function calls. During runtime, dynamic look-up is still required to find the precompiled target though, since the actual types are hidden behind interface types. For example, this scheme is effective in Lucene's ConjunctionDISI, where it joins iterators with the same document ids for conjunction queries. Without template optimization, using the DocIdSetIterator interface could lead to virtual function calls in a hot loop. However, by providing actual types via template parameters, we eliminate virtual function calls during looping.

3. Benchmark results

In this section, I present the benchmark results and analysis.

3.1. Benchmark setup

I benchmarked Java Lucene and Lucene Cyborg in the same environment as the Tantivy search benchmark game. I used Amazon Linux 2023 6.1.72-96.166.amzn2023.x86_64 as the operating system on an AWS c7i.2xlarge EC2 instance equipped with an Intel(R) Xeon(R) Platinum 8488C processor. The benchmarks were conducted using the same JDK version, OpenJDK21U-jdk_x64_linux_hotspot_21.0.2_13, across all tests.

I constructed an index using the same 7.7GB corpus provided for measuring Tantivy and Pisa in the benchmark. After indexing, I force merged all segments into a single segment then conducted the benchmark.

When compiling Lucene Cyborg C++, I used G++ version 11.4.1 20230605 (Red Hat 11.4.1-2) with the following compile flags: '-g -O3 -fPIC -march=native', along with Link-Time Optimization (LTO). Additionally, I enabled Profile Guided Optimization (PGO), which further optimizes the code based on profiling data collected during a two-phase compilation process.

In the first phase, the flag '-fprofile-generate' is added to insert logic for generating profile data, which is saved in a designated location. During execution of the built executable program in this phase, code fragments generate profile data in specified location. In the second phase of compilation, this profile data is used to inform optimization decisions.

For the PGO benchmark program, I let Lucene Cyborg C++ run against search queries used in benchmark to produce profile data, which then used for optimizing the final library to be used.

There are three types of Lucene indices tested in this benchmark, each consisting of a single Lucene segment after force merging.

1. Normal: This Lucene index was created with the DOCS_AND_FREQS_AND_POSITIONS option and includes document IDs, term frequencies, and term positions.

2. Reordered: After constructing the Normal index, document IDs were reordered to optimize search performance. The only difference between Normal and Reordered is the order of documents.

Document reordering is a technique used to represent an inverted index as a graph, aiming to assign document identifiers in a way that minimizes a specific cost function. The prominent function models the number of bits needed to store a graph or an index using a delta-encoding scheme. Given that finding the exact permutation that minimizes this cost function is an NP-Hard problem, an approximate algorithm has been proposed [Dhulipala (2016)]. This algorithm works by recursively bisecting the graph into two parts and searching for a layout that minimizes the cost function within a fixed number of trials, before merging the two parts back together. In the Lucene, this approach involves preparing two sets of document ids (left and right) at each stage and calculating a bias value for each document ids. The bias is determined using a forward index that maps document ids to term lists. It is defined as the difference between the sum of document frequencies in the left set and the sum of document frequencies in the right set [Lucene-BPIndexReorderer (2024)]. A positive bias indicates that placing the document id in the left set would be beneficial. The algorithm attempts up to 25 shuffles to move a document id from one set to the other if the move is expected to be advantageous. With this approach, it effectively clusters document ids associated with the same terms, which enhances the compression ratio of the index. Consequently, this improved compression translates into better search performance.

3. Reordered-Doc-Freq: Unlike Reordered, this index applies the same reordering algorithm used for the Reordered index to a Lucene index created with the DOCS_AND_FREQS option, which includes only document IDs and term frequencies. This index is specifically used for comparison with PISA's COUNT conjunction case, using Lucene Cyborg C++ low-level API.

For the query set, I used almost the same queries as those used in the Tantivy search benchmark game,

```
DocIdSetIterator* create(LinearAllocator* allocator,
                         std::vector<DocIdSetIterator*>& iterators) {
  ...

  DocIdSetIterator* lead1 = iterators[0];
  DocIdSetIterator* lead2 = iterators[1];

  // Given two interface types, it will find actual types for lead1 and lead2
  // then return an iterator with the template instantiated with two actual types.
  return PCO::create(lead1, lead2, &iterators, allocator);
}

DocIdSetIterator* PCO::create(
    DocIdSetIterator* param_0,
    DocIdSetIterator* param_1,
    std::vector<DocIdSetIterator*>& param_2,
    LinearAllocator* allocator) {
  // Using type index of actual types to calculate hash key.
  const auto key = ConjunctionDisiPcoCtor0TableKey({
    ((PcoTypeIndexSupport *) param_0)->get_pco_type_index(),
    ((PcoTypeIndexSupport *) param_1)->get_pco_type_index(),
  }, PRIME_HASH_KEY);

  // Find factory with calculated hash key.
  auto factory = PcoTableLookUp::look_up<
      ConjunctionDisiPcoCtor0TableKey,
      ConjunctionDisiPcoCtor0TableCell,
      CTOR_0_HASH_TABLE_SIZE>(key, &ConjunctionDisiPco_CTOR_0_TABLE[0]);

  // Return DocIdSetIterator with the template instantiated with two actual types.
  return factory(param_0, param_1, param_2, allocator);
}

DocIdSetIterator* ConjunctionDisiPco_ctor_0_factory_func_108(
    DocIdSetIterator* param_0,
    DocIdSetIterator* param_1,
    std::vector<DocIdSetIterator*>& param_2,
    LinearAllocator* allocator
) {
  // Return pre compiled optimized 'ConjunctionDisiPco' templated with two actual types.
  return allocator->allocate_object<
            ConjunctionDisiPco<Lucene90PostingsEnum<
                IndexOptions::DOCS_AND_FREQS_AND_POSITIONS>,
            Lucene90PostingsEnum<
                IndexOptions::DOCS_AND_FREQS_AND_POSITIONS>>>(
    ((Lucene90PostingsEnum<IndexOptions::DOCS_AND_FREQS_AND_POSITIONS> *) param_0),
    ((Lucene90PostingsEnum<IndexOptions::DOCS_AND_FREQS_AND_POSITIONS> *) param_1),
    param_2);
}
```

Algorithm. 1: An example of Pre-Compiled Optimization: Given two DocIdSetIterator interfaces, find a conjunction iterator that was template-instantiated with actual types. Within it, since the iterator has the required type info, it does not need to call virtual methods to join document IDs.

derived from the AOL query dataset (without any personal information). I excluded the query "to be or not to be" for both conjunction and disjunction queries after discover that additional engineering effort would be needed for BoostQuery.

3.2. Benchmark logic

In the benchmarking setup, the logic involves two processes:

1. Python driver program that is responsible for forking the engine benchmark process. It pipes search queries to the benchmark process and measures the total time taken to process them.

2. Each search engine benchmark program to execute the given commands. This process reads commands piped from standard input, processes them, prints the results to standard output, and then waits for the driver program to send the next command.

The total time taken for processing is calculated as the difference between the time when the driver process sends the request to the benchmark process and the time when it receives the results back from the benchmark process. Benchmark driver firstly warms up by feeding commands for 10 minutes per each engine without measurement.

Prime reason for having 10 minutes long of warm-up time is to make sure to reduce environment factors for the results, such as memory segment fault or Just-In-Time compilation of JVM, and measure the precise processing power of each engine.
After this warm-up time, it iterates 30 times of command measurements and pick the best result per each engine.

There are three commands that were experimented with in this paper:

1. TOP_100: This command retrieves the top 100 relevant results from the index. During the search, each engine can empowers either the WAND or Block-Max-Wand algorithm to skip irrelevant documents. The count of resulting documents may vary due to the skipping algorithm.

2. TOP_100_COUNT: Similar to TOP_100, this command also retrieves the top 100 relevant results. But it requires the exact number of matching documents. Thus skipping is not allowed in this command type

3. COUNT: This command counts the exact number of matching documents for a given query without scoring. Skipping algorithms are not applied for the sake of correctness of the count.

There are two query types that were benchmarked in this paper:

1. Conjunction: This query type finds documents that contain all of the given terms.

2. Disjunction: This query type finds documents that contain at least one of the given terms.

Below is a list of search engines included in this benchmark:

1. lucene-9.8.0-normal : Baseline Java Lucene with a normal index

2. lucene-9.8.0-reordered : Baseline Java Lucene with reordered index.

3. lucene-9.8.0-cyborg-java-normal : Lucene Cyborg with normal index.

4. lucene-9.8.0-cyborg-java-reordered : Lucene Cyborg with reordered index.

5. lucene-9.8.0-cyborg-cpp : Unlike other Lucene related engines, this benchmark program was written in C++. It exclusively supports the COUNT command for both conjunction and disjunction queries. The conjunction algorithm is adapted from PISA's implementation, while the disjunction algorithm borrows from Tantivy's. The primary goal here is to compare notable differences purely from an algorithmic perspective. Lucene includes a query rewriting step where it attempts to optimize and replace query types entirely if there is potential for computational costs reduction. For instance, it replaces a single-term disjunction query to a term query. In contrast, PISA's benchmark program directly proceeds to conjunction without a preprocessing step. For the same purpose, in the disjunction query, I adopted Tantivy's approach. Both libraries advance iterators with fixed windows to mark document IDs in a bitset and then count the marked 1s. This process continues until all iterators are exhausted. At a high level, the algorithms are similar, but Tantivy uses a window size of 4096 [Tantivy-Union (2024)] while Lucene uses a window size of 2048 [Lucene-BooleanScorer (2024)]. Additionally, in Lucene, unlike Tantivy, it maintains three heaps to manage three disjointed sets of document iterators: Tail: Iterators that are behind the current window. Lead: Target iterators belonging to the current window. Head: Iterators beyond the current window. In each iteration, Lucene combines iterators from the Tail and Lead heaps to capture the target iterators that need to be visited. The inclusion of disjunction query processing here aims to demonstrate the hypothetical lower bound of latency if it were implemented using Tantivy's algorithm.

6. tantivy-0.21 : Tantivy with version 0.21.

7. pisa-0.8 : PISA with version 0.8

3.3. Benchmark results

3.3.1. COUNT - Conjunction

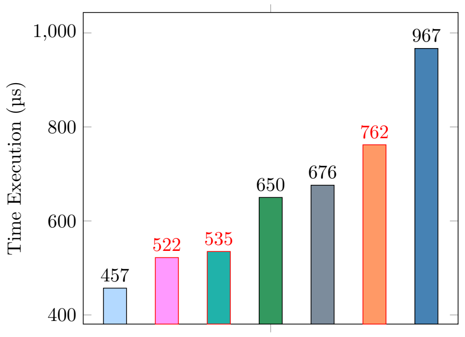

Fig 3. COUNT-Conjunction

Note that the bars with red borders represent the execution time of Lucene Cyborg.

Figure 3 presents the average time taken for conjunction queries to count matched documents. lucene-9.8.0-cyborg-java-normal demonstrates a 26.9% performance improvement compared to lucene-9.8.0-cyborg-java-normal. This trend is also evident with the reordered index, where lucene-9.8.0-cyborg-java-reordered shows a 21.4% increase in speed compared to lucene-9.8.0-reordered.

In lucene-9.8.0-cyborg-cpp, I applied PISA's benchmark logic using an index constructed with DOCS_AND_FREQ, containing only document ids and term frequencies.

In the benchmark, PISA demonstrated a 14% faster performance compared to lucene-9.8.0-cyborg-cpp. This can be a new topic for research to identify the underlying cause explaining this performance difference. Since skipping was not involved in the COUNT type queries, each iterator must have advanced an equal number of times. Therefore, this difference likely stems from the efficiency of the skip list and the decoding of document ID blocks.

The performance gap widened to 23.12% in exhaustive cases such as "+the +book +of +life", which matched 92,527 documents. PISA completed the query in 5,925 microseconds, while lucene-9.8.0-cyborg-cpp took 7,295 microseconds.

3.3.2. COUNT - Disjunction

Figure 4 displays the results for the COUNT command type with disjunction queries. In this benchmark, lucene-9.8.0-normal exibited a 32% faster performance than lucene-9.8.0-cyborg-java-normal. Similarly, lucene-9.8.0-reordered showed a 29% faster performance than lucene-9.8.0-cyborg-java-reordered.

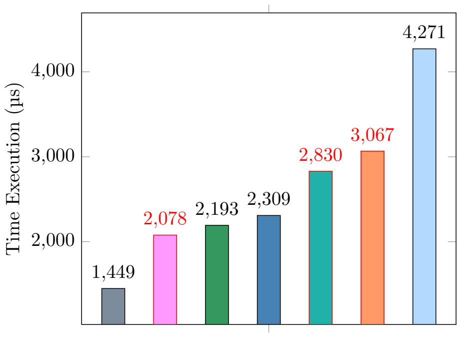

Fig 4. COUNT-Disjunction

This result is in contrast to what we observed in the COUNT-Conjunction benchmark results. Further investigation is needed to identify the underlying factors causing this difference despite using the same algorithm in both Lucene and Lucene Cyborg.

In the TOP_100 command type with disjunction query, lucene-9.8.0-cyborg-java-normal demonstrates a 17.5% faster performance compared to lucene-9.8.0-normal. However, the reasons behind this disparity are unclear at the moment. For exhaustive disjunction query types, Lucene Cyborg tends to perform slower than the baseline Lucene.

To highlight the performance differences between Tantivy and Lucene 9.8.0-cyborg-cpp, I replicated Tantivy's disjunction logic. This process involves iterating over document IDs in windows of 4096, marking the document IDs in a bitset, and finally using the "popcnt" to count the number of matched documents. The results showed that Tantivy still outperformed Lucene 9.8.0-cyborg-cpp by 43.41%.

Given that both systems process disjunctions in a similar manner without skipping irrelevant documents, the gap in performance likely stems from the way each decodes document ID blocks. In contrast to conjunctions, which depend heavily on skip lists to join document IDs, disjunctions utilize skip lists rarely. Thus, the key difference appears to be the decoding algorithm, which could explain why Tantivy exhibits lower latency despite using a similar method.

3.3.3. TOP_100 - Conjunction

Figure 5 presents the benchmark results for the TOP_100 command type with conjunction query type.

lucene-9.8.0-cyborg-java-normal shows a 21.4% faster performance compared to lucene-9.8.0-normal. This gap narrows slightly when using the reordered index, where lucene-9.8.0-cyborg-java-normal demonstrates an 18% faster performance than lucene-9.8.0-reordered.

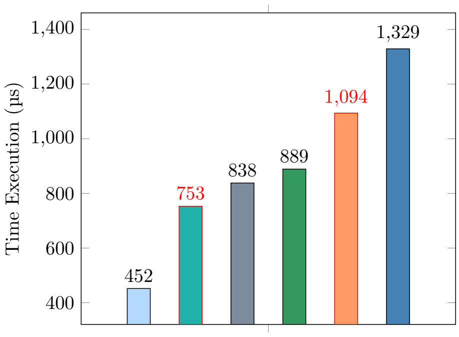

Fig 5. TOP_100-Conjunction

However, this trend does not hold true when focusing on cases having more than three terms.

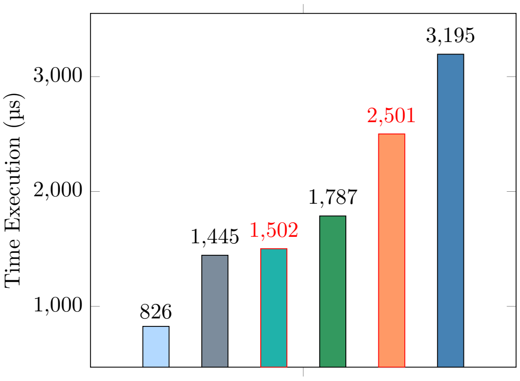

Fig 6. TOP_100-Conjunction-#terms greater than 3

When more than three terms are given, the gap widens further. In these results, lucene-9.8.0-cyborg-java-normal shows a 27.7% faster performance than lucene-9.8.0-normal. For the reordered index, lucene-9.8.0-cyborg-java-reordered is 18.9% faster performance than lucene-9.8.0-reordered.

3.3.4. TOP_100 - Disjunction

Figure 7 presents benchmark results for the COUNT command type with disjunction query type.

In these results, lucene-9.8.0-cyborg-java-normal demonstrates a 17.57% faster performance compared to lucene-9.8.0-normal. This gap widens as the number of terms increases.

Figure 8 presents benchmark results where a query consisted of more than three terms. In this benchmark, lucene-9.8.0-cyborg-java-normal exhibited a 24.79% faster performance than lucene-9.8.0-normal, while lucene-9.8.0-cyborg-java-reordered showed a 34.97% faster than lucene-9.8.0-reordered.

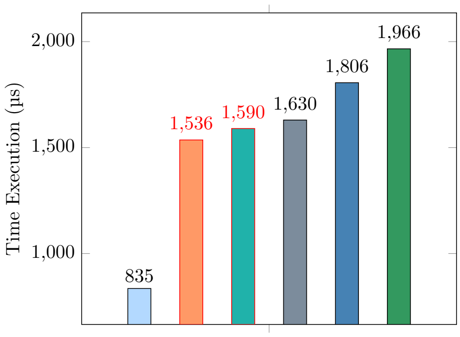

Fig 7. TOP_100-Disjunction

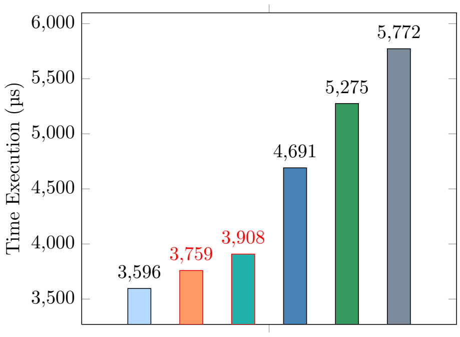

Fig 8. TOP_100-Disjunction-#terms greater than 3

3.3.5. TOP_100_COUNT - Conjunction

Figure 9 shows the benchmark results for the TOP_100_COUNT command type with conjunction query type.

In these results, the performance difference between Lucene Cyborg and Lucene appears to be narrow. lucene-9.8.0-cyborg-java-normal exhibited a 4% performance gain compared to lucene-9.8.0-normal, which was also reflected in the reordered index with a similar 4% faster performance.

Given that TOP_100_COUNT does not involve skipping and all libraries use a similar approach to achieve conjunction queries—maintaining a min heap and joining document IDs by advancing iterators—the performance gap between Lucene and Tantivy becomes noteworthy.Tantivy exhibited a performance of 810 microseconds, even faster than lucene-9.8.0-cyborg-java-reordered.

Note that I excluded PISA from TOP_100_COUNT command benchmark results after found it is actually reusing COUNT logic where it counts the number of matched documents and does not involve selecting top 100 results. [PISA-TopN (2024)]

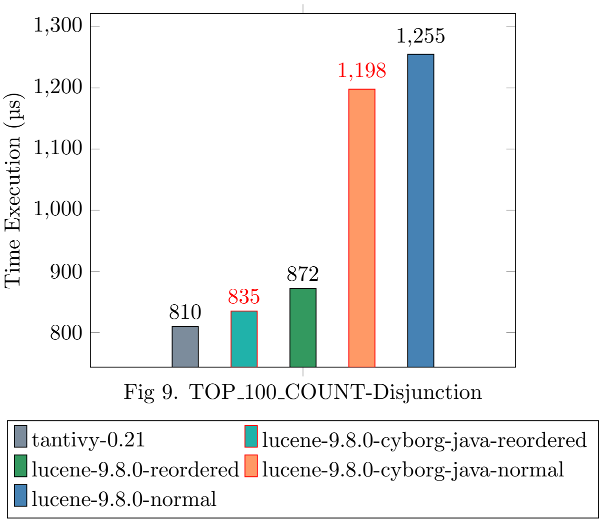

Fig 9. TOP_100_COUNT-Disjunction

3.3.6. TOP_100_COUNT - Disjunction

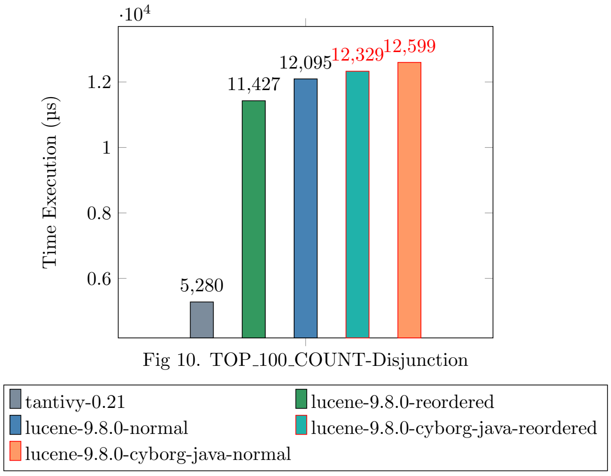

Fig 10. TOP_100_COUNT-Disjunction

Figure 10 presents the benchmark results for the TOP_100_COUNT command type with disjunction query type.

In these results, lucene-9.8.0-normal showed a 4.16% faster performance than lucene-9.8.0-cyborg-java-normal, while lucene-9.8.0-reordered showed a 7.89% faster performance than lucene-9.8.0-cyborg-java-reordered.

Since TOP_100_COUNT prohibits skipping, the complexity of the disjunction algorithm itself should be consistent between Lucene Cyborg and Lucene. The results from the COUNT command with disjunction queries indicated that baseline Lucene appears to outperform Lucene Cyborg (by 24% to 34%). Therefore, the remaining operations in TOP_100_COUNT, after subtracting the COUNT command type, should be document scoring and heap operations. It is reasonable to interpret these numbers are suggesting that Lucene Cyborg achieves better efficiency in document scoring costs and heap operations, thereby narrowing the performance gap from 24% to 4%, and from 34% down to 7%.

4. Conclusion and future works

This paper presents the core concepts of Lucene Cyborg and evaluates its performance against Java Lucene.

Lucene Cyborg demonstrated up to 27.7% better performance in conjunction search types, but exhibited relatively weaker performance in COUNT disjunction queries. Further research is needed to identify the factors affecting the performance of disjunction queries in Lucene Cyborg.

The benchmark results also highlight several intriguing topics for future research, as discussed in below.

In general, both Tantivy and PISA demonstrate superior performance in disjunction queries when skipping is disabled. Tantivy performs approximately 200% faster than the Lucene family in both TOP_100_COUNT and COUNT operations, while PISA runs faster in all benchmarks than the Lucene family.

Future experiments should focus on comparing and analyzing the factors contributing to this performance gap between the Lucene family and Tantivy/PISA.

Additionally, it would be meaningful to investigate the extent to which file format differences contribute to these performance variations. While algorithmic differences alone may impact latency in disjunction queries, it is expected that optimizations such as skip lists or other file format factors play an important role in achieving faster performance. Exploring these aspects could provide valuable insights into optimizing query execution further.

This hypothesis is drawn from the Lucene Cyborg C++ COUNT test case, which uses low-level API to execute search queries similar to PISA's conjunction benchmark logic and Tantivy's disjunction logig. However, latency results still show that PISA has a 38% performance advantage and Tantivy shows 43.41% faster execution time. Since I deliberately ensured it followed exactly the same steps as in their benchmark script, the key differences boil down to two points: 1. Efficiency in traversing in the skip list. 2. Efficiency in decoding document ID blocks.

Therefore, it would be beneficial in the future to benchmark Lucene after transplanting PISA's file format, and then compare the results to determine if there are any opportunities for improvement in the skip list and decoding algorithms.

REFERENCES

benchmarksgame. 2024, Benchmark game Java C++, 1

Broder, A. Z. 2003, Efficient query evaluation using a two-level retrieval process, 1, 426

Dhulipala, L. 2016, Compressing Graphs and Indexes with Recursive Graph Bisection, 1, 1535

Evans, J. 2006, A Scalable Concurrent malloc(3) Implementation for FreeBSD, 1

Grand, A. 2020, From MAXSCORE to Block-Max WAND: The Story of How Lucene Significantly Improved Query Evaluation Performance, 1, 20

Kim, D. 2019, Ultimate Lucene, 1

Kim, D. 2020, Ultimate Search, 1

Kim, D. 2024a, lucene-cyborg-cpp, 1

Kim, D. 2024b, lucene-cyborg-java, 1

Lucene-BKDWriter. 2024, Lucene-BKDWriter, 1

Lucene-BooleanQuery. 2024, Lucene-BooleanQUery, 1

Lucene-BooleanScorer. 2024, Lucene-BooleanScorer, 1

Lucene-BPIndexReorderer. 2024, Lucene-BPIndexReorderer, 1

Lucene-DocumentsWriterDeleteQueue. 2024, Lucene-DocumentsWriterDeleteQueue, 1

Lucene++-Issue178. 2021, Lucene++-Issue178, 1

Lucene-LZ4. 2024, Lucene-LZ4, 1

Mallia, A., Siedlaczek, M., Mackenzie, J., and Suel, T. 2020, PISA: Performant Indexes and Search for Academia, 1, 50

Masurel, P. 2016, Fast full-text search engine library written in Rust, 1

phiresky. 2021, Tantivy Finite Transducer, 1

PISA-TopN. 2024, PISA TopN, 1

Shuai Ding, T. S. 2011, Faster top-k document retrieval using block-max indexes, 1, 993

Tantivy-Union. 2024, Tantivy-Union, 1

Wikipedia-AVX. 2024, Advanced$_Vector_Extensions$, 1

Wikipedia-SIMD. 2024, Single Instruction Multiple Data, 1